UNIVERSITY OF OSLO
Department of informatics

Agent-based Extensions for the UML Profile for Service-Oriented Architectures (UPMS-A)

Master Thesis

60 credits

Ismar Slomic

1st August 2008

Acknowledgements

I would like to express my sincere thanks to my supervisors Dr. Arne-Jørgen Berre (chief scientist, SINTEF ICT) and Brian Elvesæter (research scientist, SINTEF ICT) who have given me the opportunity to learn about and acquire hands-on experience with state-of-the-art research in software engineering.

Their dedication to their work and vast technological overview has inspired me beyond my own expectations and introduced me to a whole new level of knowledge and learning. Through my affiliation with SINTEF, I have had access to a large network of scientists, engineers, research fellows and students working on more or less related research around the globe. Some of them have played an important role as discussion partners. Among these I would especially like to thank the following:

- **Dr. Øystein Haugen** (SINTEF,Norway) for sharing his expert insight into the inner workings of the UML and SOA-Pro process, and providing great motivation and guidance.

- Researchers **Christian Hahn** and **Stefan Warwaz** (DFKI Multi-agent Group, Germany) for providing me highly needed support in clarifying different parts of the case study and giving the feedbacks on my models.

- **James Odell** (Chief Technology Strategist, Oslo Software and acting chair of several OMG's groups) for giving me opportunity to be a part of the OMG Agent Group, and letting me having a word to say within the work with the AMP RFP. Your talent to present the Intelligent Agent World, in a natural and exiting way, has really encourage and helped me in digging into this field.

- **Odd Christer Brovig** (Master student and friend) for being there when i was frustrated, angry, lost and finally happy and satisfied when finished.

And last, but not least my family and all friends that have supported me all the way throughout my studies and work with thesis; thank you all!

Oslo, 01.08.2008 Ismar Slomic

Abstract

Service-Oriented Architectures are today's favorite answer to solve interoperability issues. As various kinds of systems can be used to implement Service-Oriented Architectures, the recent trend is to apply principles of Model-Driven Development by (i) modeling the Service-Oriented Architecture in an abstract manner and (ii) providing model transformations between this abstract specification and the underlying platform specific systems.

As such, Multi-Agent Systems (MASs) became very popular as both, Service-Oriented Architectures and Multi-Agent Systems, share several commonalities. In this thesis, we compare the core building blocks of Multi-Agent Systems and a proposal for a standardized UML Profile and Metamodel for Services (UPMS) requested by the Object Management Group. The major objective of this investigation is to identify if SOA-Pro–the current submission under review–offers functionalities to allow modeling of Multi-Agent Systems adequately and if not to identify what kind of functionality is missing and how this functionality can be achieved.

Interaction aspect in MASs describes how the interaction between autonomous entities or organizations take place. In our comparison we found that the ability to multicast messages is one feature that is lacking in UML Sequence Diagrams. This is an important feature or characteristic of agent interaction protocols. Here we show that SOA-Pro can easily be extended to support these kinds of functionalities.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> You may be disappointed if you fail, but you are doomed if you don't try.
>
> Beverly Sills

This chapter provides an introduction to this master thesis and begins with introducing the context of the work and the motivation for doing the research. Then we present research scope, goals and method. Document structure and relations between chapters are described in the final section.

1.1 Motivation and Background

Industry is increasingly interested in executing business processes that span multiple applications. This demands high-levels of interoperability and a flexible and adaptive business process management. The general trend in this context is to have systems assembled from a loosely coupled collection of services. These Service-Oriented Architectures (SOAs) appear to be a natural environment in which agent technology can be exploited with significant advantages. Agents deployed for IT systems generally should have the following three important properties: *Autonomous, Interactive* and *Adaptive* (see Section 3.3).

From our point of view, considering their special features, the central role that agents should play in a SOA scenario is to efficiently support distributed computing and to allow the dynamically composition of Web services. In the context of the integrated EU FP6 project ATHENA[1], it was already developed a model-driven approach for BDI[2] agents [9] based on the JACK

[1] Advanced Technologies for Interoperability of Heterogeneous Enterprise Networks and their Application

[2] Belief-Desire-Intention

[63] development environment. One of the main ideas for ATHENA was to demonstrate how models which were defined according to the *Platform Independent Metamodel for Service-Oriented Architectures* (PIM4SOA [6]) can be transformed into models that can be compiled into executable code using a metamodel definition for JACK (see [22; 30] for a detail discussion of the transformations). Furthermore, in order to use a Web service within plans of JACK agents, a second transformation that maps the concepts of a metamodel for WSDL[3] to particular concepts of the JACK metamodel (e.g. Capability) was defined. Detailed information on the model-driven framework for the integration of services into agent systems can be found in [74].

The PIM4SOA metamodel was one of the first attempts toward a metamodel for SOAs on a more abstract or platform independent level, however, its expressiveness is limited to the design of rather simple scenarios.

The Object Management Group (OMG [4]) started a standardization process for a platform independent model for services, called *UML Profile and Metamodel for Services* (UPMS). The main objectives of this new standard for services are (i) to enable interoperability and integration at the model level, (ii) to enable SOAs on existing platforms through OMG's Model-Driven Architecture (MDA) initiative, and (iii) to allow for flexible platform choices. A revised submission currently under review has been prepared that is the base for further discussions in this thesis. A brief overview on this *UML Profile and Metamodel for Services* (SOA-Pro [56]) is given in Section 3.2.4.

This master thesis is written in the context of the *Semantically-enabled Heterogeneous Service Architecture and Platforms Engineering*(SHAPE) project [14] at SINTEF ICT, Oslo, Norway. The project duration spans over two and half years, starting in the beginning of the December 2007 and ending in May 2010. SHAPE project aims to support the development and realization of enterprise systems based on a *Semantically-enabled Heterogeneous service Architecture* (SHA). SHA extends *Service-Oriented Architectures* (SOA) with *semantics* and *heterogeneous infrastructures* (Web services, Agents, Semantic Web Services, P2P and Grid) under a unified *service-oriented approach*. The SHAPE consortium consists of well-balanced combination of research, technology and application and service provider partners.

The industrial case study, which will be presented in Section 2.1, comes from the SHAPE project and will be used in our work for problem analysis and validation of our work.

[3] Web Service Description Language
[4] http://www.omg.org/

1.2 Research Goals

The general objective of this thesis is to compare the SOA-Pro proposal and underlying metamodels and profiles with key functionalities of Multi-Agent Systems (MASs), in order to investigate whether SOA-Pro offers rich and adequate functionalities to support modeling of MASs. The reasons for this comparison are twofold: Firstly, if the differences between both approaches are too rigorous, we intend to extend SOA-Pro with agent-based properties to guarantee that all strengths of MASs can actually be provided. This is of special importance if SOA models are transformed to the particular MAS models in the context of MDA. Secondly, this comparison could reveal which kind of aspects can already be modeled with SOA-Pro and which kind of aspects need to be explicitly provided in a separate *Agent Metamodel and Profile* (AMP) [54] that is currently prepared at OMG.

1.3 Scope

Service-Oriented Architecture and Multi-Agent Systems are two wide research fields, not formally standardized and still in their growing phase. Our participation in the both OMG standardization processes, UPMS and AMP, together with case study has gained us knowledge about these two fields, and also helped us identifying requirements for the comparison of SOA-Pro. These are presented in Section 2.5.

Covering all these needs and belonging solutions, simply is not possible within the work with this master thesis. In fact, we had to choose only to concentrate on the interaction aspects in the MASs, which describes how the interaction between autonomous entities or organizations take place. Reason for choosing this aspect, among several others, is because of its importance with respect to the case study and its requirements.

1.4 Research Method

In this thesis we apply a research method that is a combination of the Technology Research, presented by Solheim and Stølen in [33], and the ACM's taxonomy of computer science presented in [13]. This kind of research methods concerns the development of a *new artefact* or *improvement* of an existing one.

1.4.1 Method

According to Solheim and Stølen [33], *Technology Research* is an iterative process consisting of the main steps as illustrated in Figure 1.1:

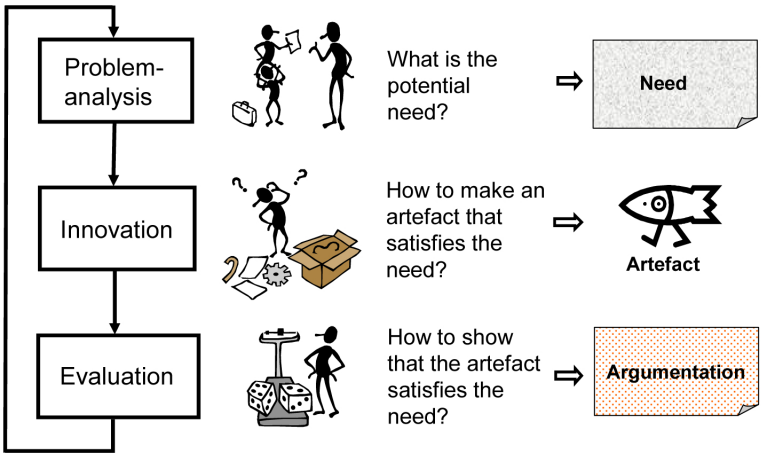


Figure 1.1: *Method for Technology Research - main steps, from [33].*

- **Problem analysis** - find a problem to which a solution is needed by interacting with possible users and other stakeholders.
- **Innovation** - construct an artefact that satisfies the potential need. The overall hypothesis is that the artefact satisfies this need.
- **Evaluation** - based on the potential need, formulate predictions about the artefact and checks whether these predictions come true. If the predictions turn out to be correct, it can be argued that the artefact solves the identified problem.

The results are validated by performing test-cases, which will either strengthen or weaken the hypotheses. This process may be repeated several times, depending on the result of the validation.

What distinguishes technology research from technology development is the artefact's representation of *new knowledge* of some general interest. In order to decide whether an activity is technology research or technology development, we need to answer the following three questions:

1. Does the new artefact represent new knowledge?
2. Is the new knowledge of interest to others?
3. Is the new knowledge and results documented in a way that enables validation by others?

In Figure 1.2, the research method is illustrated in the context of this thesis. The three steps are elaborated further in subsections 1.4.2, 1.4.3, and 1.4.4.

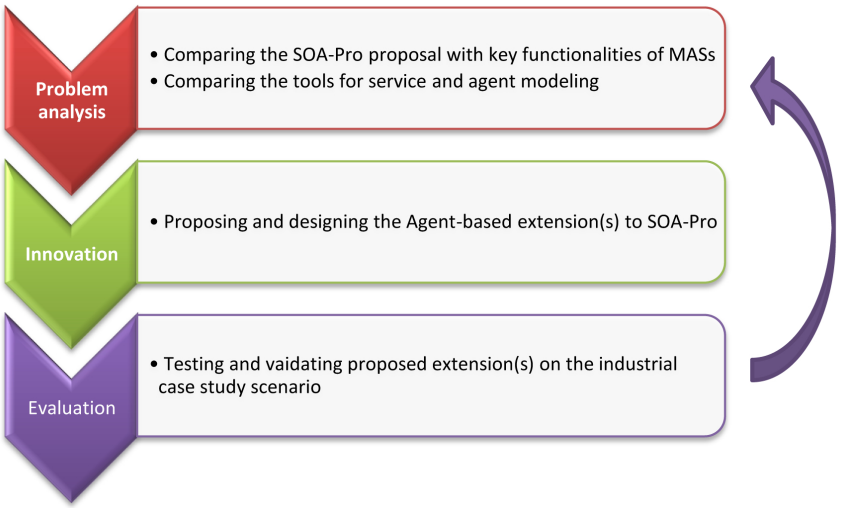


Figure 1.2: *Illustration of the research method in context of this thesis.*

1.4.2 Problem Analysis

Initial step of our work was comparing the SOA-Pro proposal with the core concepts of MASs, in order to identify the similarities and needed areas where extensions were needed. We choose to compare SOA-Pro with the agent aspects: agent aspect, collaboration aspect, role aspect, interaction aspect, behavioral aspect and mental aspect (see Section 2.5).

Since the interaction aspect was central in the solution of our case study, and we identified more problem areas than we were able to find solutions for in the work of this thesis, we choose to focus further on the interaction aspect.

In the context of the SHAPE project, where one of the main objectives is to develop open-source tools, in order to support enterprise systems based on a SHA, we wanted also to compare tools for modeling service- and agent-oriented systems. This would help us identify the most feasible state of the art tool covering our requirement (see Section 2.6 for requirements and evaluation).

1.4.3 Innovation

When the problems were allocated in previous step, we analysed further the interaction aspects within SOA-Pro at this step of the research method. The aim was to identify solutions (artefact) to the identified problems.

Based on the identified problems, and discussions in Chapter 2, requirements were presented for interaction aspect that needed to be satisfied by the resulting artefact. However, there have been situations during this process

that required that we took a step back, and analysed further, as additional problems arise. The artefact in our work is presented in Chapter 4.

1.4.4 Evaluation

Validating the results is very important in order to confirm that result actually solves the identified problem. Thus we must validate that the extensions to the SOA-Pro actually improves the expressiveness for agent interaction protocols. The basis for the validation was to create predictions regarding the hypotheses based on the interaction aspect extensions (see Section 2.2 and 2.3). Furthermore, the extensions were tested on the case study scenario (see Chapter 5).

Thus; the prediction tried to predict how the proposed extension would do improvement to SOA-Pro proposal, in order to specify interaction protocols, which is important within MASs. The results were validated (see Chapter 6) in according to the evaluation criterions. If the artefacts proved to be insufficient, we needed to take a step back to make improvements, or it might be necessary to go back to the problem analysis with the newly gained knowledge to analyse further.

1.5 Structure of this Thesis

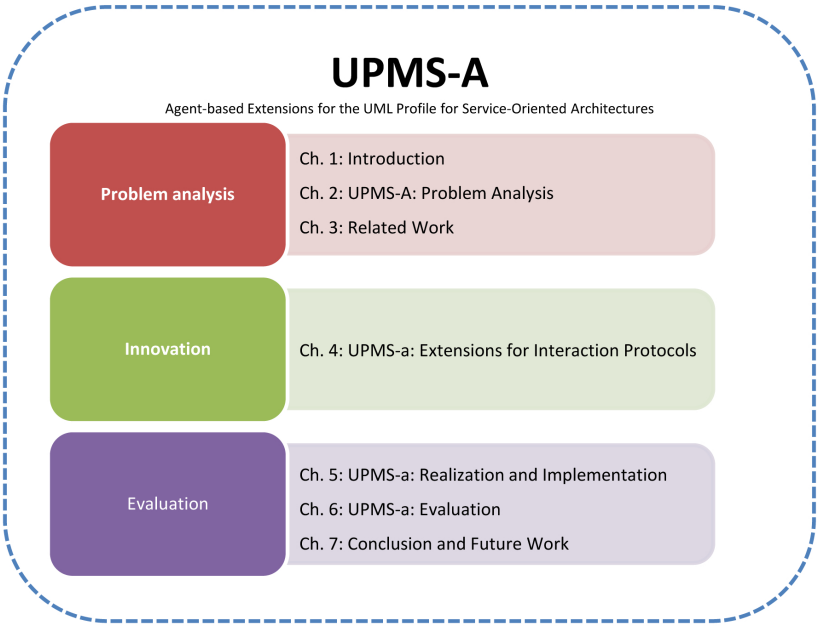


Figure 1.3: *Thesis structure overview.*

The structure of this thesis is basically divided into three main parts, following the research method steps, as shown in Figure 1.3.

In the chapters of the **Problem Analysis** part, we present the background and context of our work, with problem analysis, case study description and related work which the reader needs to understand in order to follow our work. Here we give briefly introduction to Model-Driven Development (MDD), SOA, Agents and MASs, and introduce the reader to two commonly used interaction protocols.

The **Innovation** part consists of only one chapter and that is proposed extensions for interaction protocols. Note that we are using the acronym UPMS-**A** in the name of Chapter 2, while UPMS-**a** in Chapters 4, 5 and 6. With this notation we want to illustrate that in chapter with problem analysis (2), we introduce several aspects where SOA-Pro could and should be extended in order to support modeling of MASs. But we only propose solution on *one* of these aspects, and therefore small-case letter **a**.

Finally, the third part **Evaluation** includes chapter that present realization and implementation of our extension and furthermore evaluates extensions with respect to the evaluation criterions that we presented in the chapter with problem analysis. Last chapter concludes our work and gives several future work ideas.

Chapter 2

UPMS-A: Problem Analysis

> It isn't that they can't see the solution. It is that they can't see the problem.
>
> Gilbert Keith Chesterton

In this chapter we will present the problem analysis that we did in the early stage of our work, as in accordance with the research method. First, we start with presenting the case study in Section 2.1, which is the supply chain in steel productions. Section 2.2 depicts the hypothesis while Section 2.3 the success criterions (predictions). Requirements for the interaction protocols are described in Section 2.4. In Section 2.5 we name the core building blocks of MASs and discuss in which manner the particular aspect can be expressed using SOA-Pro Finally, Section 2.6 presents requirements and evaluation of the state of the art modeling tools.

2.1 Case Study: Supply Chain in Steel Production

Saarstahl AG[1], with its locations in Völklingen, Burbach and Neunkirchen along with Roheisengesellschaft Saar in Dillingen (Saarstahl and Dillinger Hütte each with 50%) is one of the most important manufacturers of long products in the world. The company is recognized as having a high level of competence in the field of steel production and further processing.

Saarstahl AG is a German steel manufacturing company with global presence on the steel production market. In particular, Saarstahl AG specializes in the production of wire rod, steel bars, and semi-finished products of various grades as well as constructional steel and broad flanged beams. The

[1] http://www.saarstahl.com/

product range also includes open die forgings. These products are important preliminary products, both today and for the future, for the automotive industry and its suppliers, the construction industry, power industry engineering, the aerospace industry, general mechanical engineering, and other steel processing branches.

The production of steel normally is the first phase of most Supply Chains in different areas. Steel manufacturing companies are strongly affected by bull whip effect. Due to the irregular nature of incoming orders and the frequently changing customer requirements on accepted orders, making the right decision at a certain stage can make the difference between earning or loosing. In order to keep a competitive position on the market, it is important to improve operational efficiency. To achieve this, flexible planning and scheduling systems, capable of handling considerable amounts of data, are needed. Existing systems are commonly centralized decision making approaches, mostly data driven and often not modeling the business processes conveniently.

Figure 2.1: *Steel production in Saarstahl AG.*

In the past, Saarstahl has made great efforts to deal with the planning and scheduling problems along its production chain. These are different from supply chains like discrete productions in the automotive cluster. Steel production is a disassembling, continuous process starting from hot metal which is almost anytime similar and resulting in a vast number of different products. Time restrictions are more important than in other production chains, since certain processes cannot be interrupted. For instance, hot metal leaving the blast furnace factory must be transformed and casted into steel billets within a certain time, because of the temperature it has to keep during the process before it goes from liquid to solid.

Given a working plan, the system schedules the execution of each order along the production chain. It monitors production on a rough (weeks) and

detailed (days and hours) level, and executes an online detailed planning and scheduling for the different manufacturing phases. It has to detect problems in the production and handle them in order to return to normal production. The rough working plan for each manufacturing phase is calculated on demand, before final order commitment. Depending on delivery date, order size and vertical integration certain capacities at specified aggregates have to be roughly allocated.

Usual orders to Saarstahl vary between five to several hundreds of tons. Batch sizes on each manufacturing level are fixed or limited, hence, orders have to be grouped together in process units on each stage with local constraints to keep. For instance, inside the steel work, a production unit is called *heat* with fixed size of **160 t**. The orders covered by a heat have to be of same quality, same casting format, and should have the same calculated processing step date.

2.1.1 Use case "Creation and Optimization of Heats and Sequences"

In higher planning levels (sales), the global production capacities for the different production phases are booked. After that, the planning process continues by planning at lower levels. In the case of the creation and optimisation of heats and sequences, the global planning level provides the lower level with a set of orders. This set consists normally of about 3500 *order positions* of different sizes, deadlines, qualities, and further restrictions related to each order position. These positions have to be mapped into *heats* of a fixed size of 160t (see Figure 2.2).

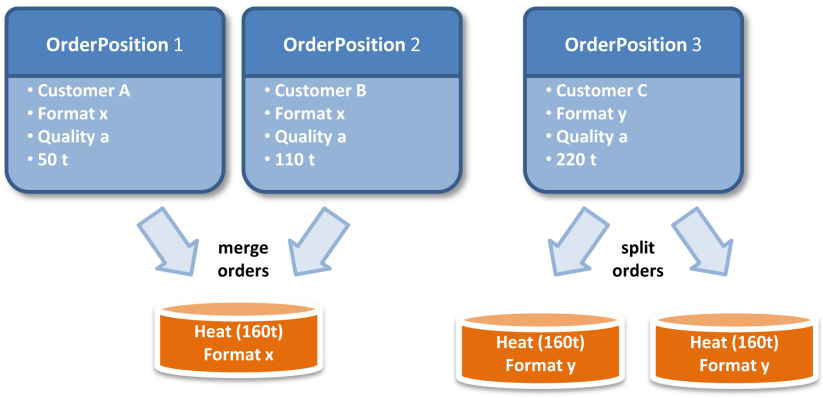


Figure 2.2: *Assignment of order positions to heats.*

Aggregates are the production steps of the steelwork. Each aggregate has certain capabilities. In order to produce a specific product, the liquid steel has to be processed by suitable aggregates. Because of the continuous

process, heats with equal quality and similar time constraints have to be grouped into *sequences* to reduce the setup times and down times of the aggregates (see Figure 2.3). After a sequence of heats has been processed by an aggregate, the aggregate requires a certain setup time before the next sequence can be processed. Therefore, the length of every sequence has to be maximised to reduce production costs.

Heat and sequence creation is divided in two levels:

1. First, an initial heat creation is calculated. In this phase, the order's deadline is the major criterion. The aim is to minimise the number of heats to optimise order throughput and minimise costs.

2. Secondly, sequences are created. Maximising a sequence's length means to minimise down times of the continuous casting aggregate and hence optimising the aggregate's throughput.

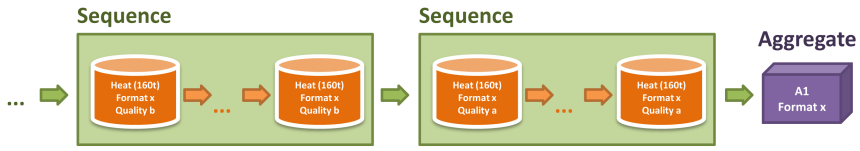


Figure 2.3: *Aggregates and sequences.*

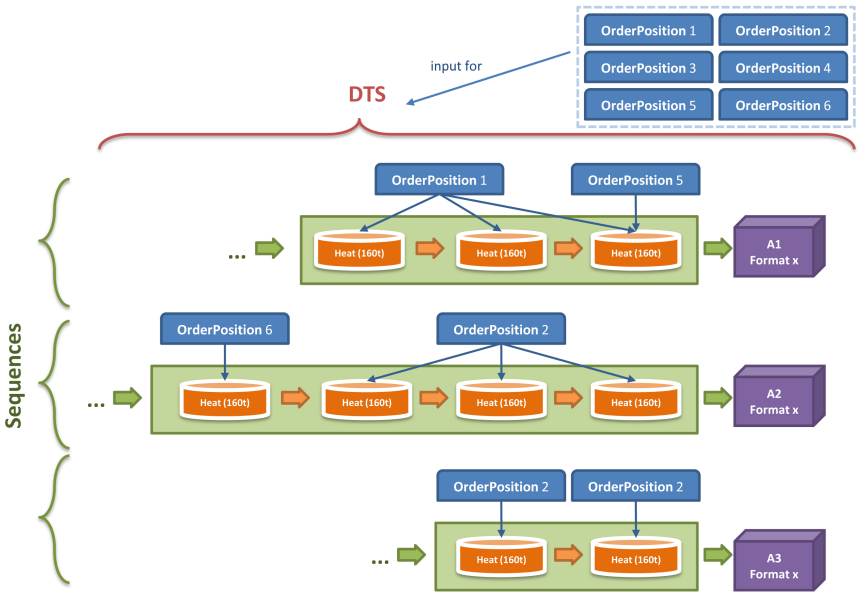


Figure 2.4: *Example daily target schedule (DTS).*

The result of these two phases is a base for the creation of a *daily target schedule* (DTS) as a presetting for the production inside the melting shop.

This DTS consists of a partial ordered set of sequences for the continuous castings inside the steelwork. Each sequence consists of a total ordered set of heats (see Figure 2.4).

2.1.2 Use Case Challenges

As mentioned, the first phase is the creation of heats as batch size for the steelwork. Input for this process is the order backlog **R** of order positions which still have to be melted, average |R| 4000.

Major criterion is the latest possible manufacturing completion date of the steelwork. Also, other restrictions are mandatory, these are:

Steel Grade: different order positions may not be inside the same heat

Casting Dimension: the formats of all different positions must be equal in order to be inside the same heat

Continuous Casting Aggregate: order positions are mapped to determined aggregates

Subject to these restrictions, R has to be partitioned into several subsets so that the union of all subsets defines the whole original set R. A heuristic is used to create the heats as initial solution for the second phase.

At first, each R_j is sorted lexicographically by delivery date and order size. Some orders have capacity greater than one heat, hence at least one complete heat is allocated by such orders and the orders have to be separated into several parts. Because of different lengths of billets (limited by certain legacy systems) and order sizes further orders need to be separated and distributed on more than one heat. Some customers demand their materials of a single heat. This is also taken into account in this first step. Now, new heats are created anytime a certain order does not fit completely into an instantiated heat. The system should not separate and distribute orders if not necessary. Hence, according to urgency and size, heats are created until a particular minimum is reached. Secondly, all orders which have not been assigned yet have to be mapped to existing heats or probably new heats have to be created. In this step, a score function is used by each order to determine how worthwhile it is to get into a certain heat. Hence, the overall costs are minimised. The score must be inside an user defined range. Since it is possible, that certain orders might not be assigned according to this range, in the next step, the “best" score is criterion. The first phase is closed by a plausibility check on the filling degree. As a result, heats with a filling degree greater than 95% have to be received.

In the second phase sequences have to be created and its compositions have to be optimised. The planning department chooses a certain number of heats as a sequence. This set consists of a set order positions. Since

a sequence is created for production, the major criterion “latest possible manufacturing completion date" of the first phase has become irrelevant. Other, new restrictions are mandatory, these are:

- The filling degree of a heat must be kept inside a certain tolerance range
- Number of semi finished products lengths is limited to 4
- Each order position might not be separated on more that three heats

Most important criterion in this scenario is the degree of degassing. Certain orders need to be degassed for reasons of homogeneity. During the first phase, this criterion has not been taken into account since it is counterproductive to the latest possible manufacturing completion date. Orders which need a degassing are evenly distributed on each R_j in the initial solution. Purpose of the second phase is to group all orders which need a degassing into equal heats. So, the number of degassed heats and therefore production costs will be minimised.

The former approach at Saarstahl was to solve it manually. An employee of the planning department chose a certain subset - the length of the correspondent sequence - and tried to exchange order positions between the heats in order to optimise the number of degassed heats. Because of the complexity of the problem and the fact that this has to be done for almost every sequence of DTS, an automated solution was needed.

The presented approach uses Simulated Trading (see Section 3.4.3) to solve it. It is an improvement mechanism starting from any initial solution with random heats as generated during the heat creation. By successively “selling" and “buying" certain order positions each heat tries to optimise its composition of order positions. Objective is to achieve a new assignment of the already accepted order positions to the heats with an optimised cost. The trading goes over several rounds. In each cycle the heat agents submit one offer to sell or buy an order position. At the end of each round a trading agent tries to match the sell and buy offers.

This is a special kind of hill-climbing algorithm, which can be interrupted anytime to pick the best solution found. This has to be done with all created subsets in parallel which are the number of sequences in DTS. The protocol is depicted in the Section 3.4.3 (Figure 3.22).

The Saarstahl case is a proof of concept for designing the main processes within the supply chain based on the results of our thesis work. For the “Creation and Optimization of Heats and Sequences" scenario, the following issues are of main importance.

- Which parts need to be modelled and how to model particular parts?
- How to simplify the orchestration by using process modeling and services?

- How to integrate planning and scheduling algorithms into SOA? How to integrate existing legacy systems? For Saarstahl, it is fundamental that existing systems (e.g. data bases) can be re-used within the SOA to maintain the high product quality.

2.2 Hypothesis

While in the previous part of the problem analysis we described a case study, recognized the problems within its domain and proposed a solution, in here we state the hypothesis that this thesis needs to address:

H1 Proposed extensions will make SOA-Pro and UML suitable to express agent interaction protocols.

2.3 Definition of Success Criteria

Since the hypothesis, H1, is stated, and the requirements for its main parts listed, the next step would be to define the success criteria by which we will test the hypothesis. The success criteria will focus on the expectation of H1, meaning the improvement of the SOA-Pro in order to better support agent interaction protocols. Through the success criteria we define the desired effect that UPMS-a will provide. They are presented as predictions, and will be evaluated in Chapter 6. The evaluation will validate the hypothesis H1.

Success criterion 1 A suitable extension of SOA-Pro will make it possible to express multicast of messages between participants in agent interaction protocols.

Success criterion 2 A suitable extension will make it possible to group participants in groups in order to express which group of participants are receiving or sending particular message.

Success criterion 3 A suitable extension will only introduce or extend the SOA-Pro proposal where needed, and use as much as possible existing concepts in SOA-Pro or UML.

Success criterion 4 A suitable extension will make it clear how to use timer concepts within SOA-Pro in order to express deadlines in interaction protocols.

In order to investigate further the hypothesis, we will define extension requirements for interaction protocols, which will be used to evaluate SOA-Pro in Section 2.5.4.

2.4 Requirements for Interaction Protocols in SOA-Pro

Since Simulated Trading Protocol (STP) is central in solution to the use case in our case study, and we believe that this protocol represent features commonly needed within agent interaction protocols, it is required that SOA-Pro supports for modeling these protocols.

In order to support agent interaction protocols, following features are needed:

1. Multi-receiving and sending of messages. For instance the Initiator is sending several *call for proposal* (cfp) messages in one step to potential participants, and is receiving one or many responses in return.

2. Grouping of the participants. It is very important to be able to express which group of participants is receiving or sending the messages.

3. Iterating multicast messages from and to particular participant groups, in order to express possible message exchange between Initiator and Participant

4. Defining timer concepts in order to express deadlines and prevent the possible deadlock when the Initiator waits for responses from Participants.

	Interaction Protocols Requirement
IPR1	Multi-receiving and sending of messages
IPR2	Grouping of participants
IPR3	Iterating multicast messages from participant groups
IPR4	Defining timer concepts

Table 2.1: *Summarized requirements for interaction protocols.*

2.5 Comparing SOA-Pro With MASs Aspects

In general, agents can be software agents, hardware agents, firmware agents, robotic agents, human agents, and so on. While software developers naturally think of IT systems as being constructed of only software agents, a combination of agent mechanisms might in fact be used from shop-floor manufacturing to warfare systems.

These properties are mainly covered by a set of core building blocks, each focusing on different viewpoints of MASs. Even if these aspects do not directly appear in SOA-Pro, we can relate them to concepts used in

SOA-Pro. In the following, we name the core building blocks of MASs and discuss in which manner the particular aspect can be expressed using SOA-Pro. However, our focus here is on the Interaction Aspect described in Section 2.5.4, and we will evaluate it in according to the requirement we specified in previous section.

2.5.1 Agent aspect

Agent aspect describes single autonomous entities and the capabilities each can possess to solve tasks within an agent system. In SOA-Pro, the metaclass *Agent* describes a set of agent instances that provides particular service capabilities. As the metaclass Agent inherits from the metaclass *Participant* (see Figure 3.5), an Agent can be considered as entity providing services (i.e. capabilities). This property nicely corresponds to the manner in which agents and capabilities are linked in MASs. Agents in SOA-Pro are specialized because they have their own thread of control or life cycle. Another way to think of agents is that they are *active participants* in a SOA system. Participants are *Components* whose capabilities and needs are static. In contrast, Agents are Participants whose needs and capabilities may change over time. A Participant represents some concrete Component that provides and/or consumes services and is considered as an active class. However, SOA-Pro restricts the Participant's classifier behavior to that of a constructor, not something that is intended to be long-running, or represent an active life cycle.

2.5.2 Collaboration aspect

Collaboration aspect describes how single autonomous entities collaborate within MASs and how complex organizational structures can be defined. In SOA-Pro, a *Contract* indicates roles interacting within this part and how messages are exchanged between these parties, which is mainly done through UML Sequence Diagrams. In SOA-Pro, a contract fulfillment (*CollaborationUse*) indicates which roles are interacting (i.e., which parts they play) in the contract. The concept of a *ServiceContract* can be used to model simple collaborations in MASs. However, social units like organizations and groups that are formed by agents during run-time to take advantage of the synergies of its members, resulting in an entity that enables products and processes that are not possible from any single individual are out of scope of SOA-Pro.

2.5.3 Role aspect

Role aspect in MASs covers feasible specializations and how they could be related to each role type. In SOA-Pro, the concept of a role is especially used in the context of *ServiceContracts*. Like in MASs, the role type indicates which responsibilities the particular entity takes on. However, in MASs,

several different notions of the term role can be considered. Often, beside the more domain-related concept of a role, especially in collaborations between agents, social roles are typically used to express the power relationship between participating entities. Furthermore, in MAS, roles are considered as first class entities. Like agents, that can have access to particular capabilities, behaviors, and resources. These more complex characteristics are not part of SOA-Pro, and can even be hardly modeled using pure UML.

2.5.4 Interaction aspect

Interaction aspect in MASs describes how the interaction between autonomous entities or organizations take place. Each interaction specification includes both the entities involved and the order which messages are exchanged between them in a protocol-like manner. In SOA-Pro, *ServiceContracts* are the place where interactions are defined. Like agent interaction protocols, a services contract takes a role centered view of the business requirements which makes it easier to bridge the gap between the process requirements and message exchange. Furthermore, a Contract can have an owned behavior which in most situations would be a UML Sequence Diagram. However, the ability to multicast messages is one feature that is lacking in UML Sequence Diagrams which is an important feature or characteristic of agent interaction protocols.

IPR1 As we mentioned above, UML Sequence Diagrams are often used to express interaction between parties. However, UML Sequence Diagram does only support scenarios where messages are sent to only single entity and is lacking in supporting the multi-send and receive of messages.

IPR2 This requirement are partly fulfilled by Sequenced Diagrams, since it is possible to have one lifeline for each of the typical group a participant may be in. In this it is possible to visualize more directly that there are several participants. We are not able, however, to describe the multiplicities of each of the groups.

IPR3 Since we already have pointed that multicast is lacking feature in UML Sequence Diagrams, this feature has never been needed and therefore this requirement is not fulfilled.

IPR4 SOA-Pro does not mention timer concepts at all. Although UML provide SimpleTime model in the UML Superstructure, it is rarely used and many designers are not even aware of that it exists.

0: Requirement is not fulfilled **1:** Requirement is partly fulfilled **2:** Requirement is fulfilled

	Evaluation	**Score**
IPR1	Multi-receiving and sending of messages	0
IPR2	Grouping of participants	1
IPR3	Iterating multicast messages from participant groups	0
IPR4	Defining timer concepts	1

Table 2.2: *Summarized interaction protocol evaluation of SOA-Pro*

2.5.5 Behavioral aspect

Behavioral aspect in MASs describes how plans are composed by complex control structures and simple atomic tasks such as sending a message. In SOA-Pro, a *ServiceInterface* is a *BehavioredClassifier* and can thus contain ownedBehaviors that can be represented by UML 2 *Behaviors* in the form of an *Interaction, Activity, StateMachine, ProtocolStateMachine*, or *OpaqueBehavior*.

2.5.6 Mental aspect

Mental aspect in MASs defines concepts like Beliefs, Intentions, and Goals. Such concepts are neither part of the metamodel of SOA-Pro nor part of the profile. However, these concepts are very useful when designing MASs based on a BDI architecture.

2.6 Tool Requirements & Evaluation

In order to solve some of the problems in our case study, we need sufficient UML Tool which we can use for service and agent modeling. Since SOA-Pro provides description of its UML Profile, we want to be able to define and apply UML Profiles, in addition to other requirement as described in next section.

2.6.1 Requirements

The tool shall:

1. be implemented in an open source technology.
2. be metamodel-based. The tool shall not only base on the UML 2 metamodel, but also check for correctness of model elements against the metamodel.
3. give support for defining stereotypes which extends elements of UML 2. Afterwards, the profile shall be easy to apply to models and model elements, and be displayed graphically.

4. represent models as common XMI Schemas to facilitate interchange of semantic models, according to MDA approach.
5. graphical support for UML 2 Class Diagrams.
6. graphical support for UML 2 Sequence Diagrams with features as; combined fragments (alt,opt,break..) and messages representing operations or signals.
7. graphical support for UML 2 Component Diagrams with features as; components, ports and owned elements (activity or sequence diagrams describing behaviour).
8. graphical support for UML 2 Activity Diagrams.
9. have a graphical interface that could design and manipulate visual models in different views.
10. be easy to use and stay in a stable condition so the user doesn't experience lot of bugs and unexpected errors.

	Tool requirement
TR1	Open-Source technology
TR2	Metamodel-based
TR3	UML Profiles
TR4	XMI Schemas
TR5	Class Diagram
TR6	Sequence Diagram
TR7	Component Diagram
TR8	Activity Diagram
TR9	Multiple-view support
TR10	Usability & Stability

Table 2.3: *Tool requirements.*

2.6.2 Evaluation

Four modeling tools were evaluated, according to requirements specified in previous section, and mainly just two of them could be useful in our work. Summarized comparison of tools is collected in Table 2.4.

UML 2 Tools v0.7.1 [2] is a set of GMF-based (The Eclipse Graphical Modeling Framework) editors for viewing and editing UML models, and is a part of the *Model Development Tools* (MDT) Eclipse project.

[2] http://www.eclipse.org/mdt

UML 2 Tool is in incubation phase now and there is huge lack of functionality and UML support compared with other tools. But we believe that this tool will be very valuable in the meaning of the MDA approach in the future work since it is open-source, built upon Eclipse platform and represent models as common XMI Schemas which makes the interchange of models between tools possible.

Objecteering Enterprise Edition v6.1 (Obj.) [3] claims to be a tool for complete coverage of Model-Driven Development, from goal analysis right through to coding and testing. The vendor, SOFTEAM[4] offers Free Edition of the modeling tool, but in order to use UML Profiles you are required to upgrade and pay for commercial Enterprise Edition. Objecteering seems to be sufficient for simple UML modeling, but when it comes to defining and applying UML Profiles, it is lacking in usability, functionality and UML 2 compatibility. We should expect more of an commercial product and therefore it almost gets the same score as UML 2 Tools.

Papyrus v1.9.0 (Pap.)[5] is a dedicated tool for modeling within UML 2. This open source tool is based on the Eclipse environment and key features are (i) compatibility with UML 2, (ii) development support for UML Profiles and (ii) code generation from models. It covers most of our needs and is ranked as number 2 in our evaluation.

IBM Rational Software Modeler v7.0.5(RSM) [6] is the most complete tool of these four we have evaluated. It covers all of our requirements and even more. Only drawback is that it is commercial. However, through IBM Academic Initiative[7], it is possible to gain access to software, hardware, training and other benefits, with no cost and this has been the main reason why we choose this tool instead of Papyrus.

[3] http://www.objecteering.com/
[4] http://www.softeam.fr
[6] http://www.ibm.com/developerworks/rational/products/rsm/
[7] http://www.ibm.com/university/academicinitiative

0: Requirement is not fulfilled

1: Requirement is partly fulfilled

2: Requirement is fulfilled

	Requirements	**Obj.**	**Pap.**	**UML 2 Tools**	**RSM**
TR1	Open-Source technology	0	2	2	0
TR2	Metamodel-based	1	2	1	2
TR3	UML Profiles	1	2	0	2
TR4	XMI Schemas	0	2	2	2
TR5	Class Diagram	2	2	1	2
TR6	Sequence Diagram	1	2	0	2
TR7	Component Diagram	1	1	1	2
TR8	Activity Diagram	2	2	1	2
TR9	Multiple-view support	0	0	0	2
TR10	Usability & Stability	1	1	0	2
	Score	**9**	**16**	**8**	**18**

Table 2.4: *Summarized tools evaluation.*

Chapter 3

Related Work

> Not to know what has been transacted in former times is to be always a child. If no use is made of the labors of past ages, the world must remain always in the infancy of knowledge.
>
> Marcus Tullius Cicero

This chapter introduces the technologies, methodologies and concepts this thesis are based on. Section 3.1 presents the foundations of the Model-Driven Development (MDD). Section 3.2 explain the basics of the Service-Oriented Architecture (SOA), which provides an technology independent architecture for the integration of business processes. We assume that reader has already some knowledge about this topic and will not cover all the details. Here we also introduce core concepts of UML Profile and Metamodel for Services (SOA-Pro).

Section 3.3 introduce the Agent concepts. Questions like "what is an Intelligent Agent?" and "why are Agents useful?", are answered in this section. We also explain the basics of Multi-Agent Systems (MAS) and Agent-Oriented Software Engineering (AOSE). Finally, Section 3.4 explain how we can optimize supply-chain process in steel production by interaction with use of two well-known interaction protocols in MAS.

The last two sections about Agents are the biggest part in this chapter. The reason why using so much space is that we believe this is still an undiscovered field for most readers and could somehow be difficult to relate to. At the same time we feel that it's important to have the basic understanding to be able to follow further discussions in this thesis.

3.1 Model-Driven Development (MDD)

Model-Driven Development (MDD) is a software engineering approach with particular focus on models, automation and code generation. The difference to traditional software development is that MDD proposes to leverage models to generate the specified software system. MDD aims to leverage models for automatically generating applications with appropriate code generation techniques and templates. Models are already used to specify software systems, but unfortunately these models mostly serve only for the purpose of documentation and comprehending the system. Changing this fact by using these existent models to generate the application, software development can easily be automated. By automatic code generation, the quality of an application can be increased, due to the fact that code is produced according to a certain structure, scheme or rules. In this way the generated code will precisely match the models. Further on this road the evolution could lead to the fact that modeling languages replace the implementation languages, just like the way third-generation languages replaced the assembly languages through the introduction of compilers.

Two currently dominant approaches to MDD are *Model-Driven Architecture* (MDA) [51] and *Software Factories* (SF) [27]. We will first start with giving a short introduction in these two approaches and then at the end compare them with each other in order to discover their strength and weakness.

3.1.1 Model-Driven Architecture (MDA)

Impelled from the idea that models are vital and necessary to handle complexity in software development, Model-Driven Architecture (MDA) [51] specifies a process for creating models.

The *Unified-Modeling Language* (UML) [58] is the proposed modeling language for MDA. According to [60, page 11-19], UML's architecture is represented by a four layer hierarchy as shown in Figure 3.1. On the top of this hierarchy is the meta-metamodel layer (M3), also known as the *Meta-Object Facility* (MOF) [59] layer, followed by the metamodel layer (M2) representing the UML. The third layer is model layer (M1), where the actual modeling process takes place. The architectural hierarchy bottoms out at the data layer (M0), which contains the run-time instances of a model.

Furthermore, UML provides extension mechanisms (**Stereotypes, Tagged Values** and **Constraints**) to extend or specialize itself for specific purposes and domain. A collection of Stereotypes, Tagged Values and Constraints is called a *UML Profile* (see Figure 3.5 for example). Profiles are necessary and needed in the context of MDA, because they facilitate code generation.

Examples of some UML Profiles are *UML Testing Profile* (see Section 4.5.3, page 91), *OMG Systems Modeling Language* (SysML) and *UML Profile for*

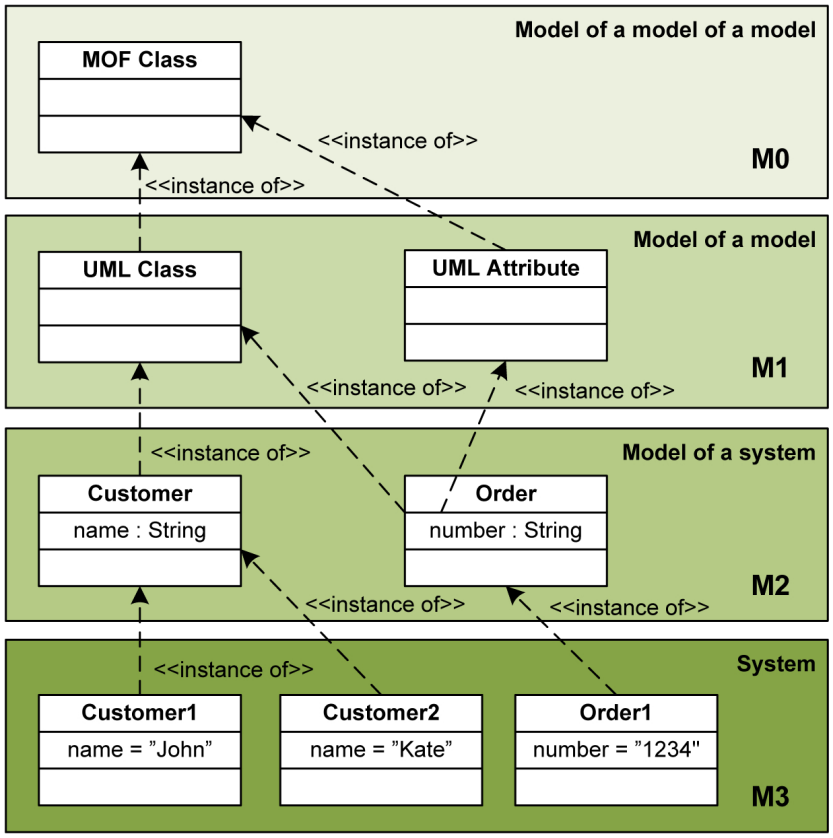


Figure 3.1: *Overview of the four modeling metalevels, defined by OMG.*

Corba. These, and several other profile specification can be downloaded from OMG's catalog of UML Profile specifications [55].

3.1.1.1 Basic MDA Concepts

Platform independency is an important part of MDA. Particularly due to this aspect, the software system is supposed to be modeled in three major steps. Starting with the *Computational Independent Model* (CIM) to present a high-level overview of the software system, continuing with the *Platform Independent Model* (PIM) encapsulating the entire software specification, apart from the any platform specific aspects, the modeling process finishes with the enhancement of the PIM to a *Platform Specific Model* (PSM). The PSM binds the software system to a specific platform.

3.1.1.2 Levels of Abstraction

Another important aspect of MDA is the transformation between model. In order to achieve the independence from the software application platform,

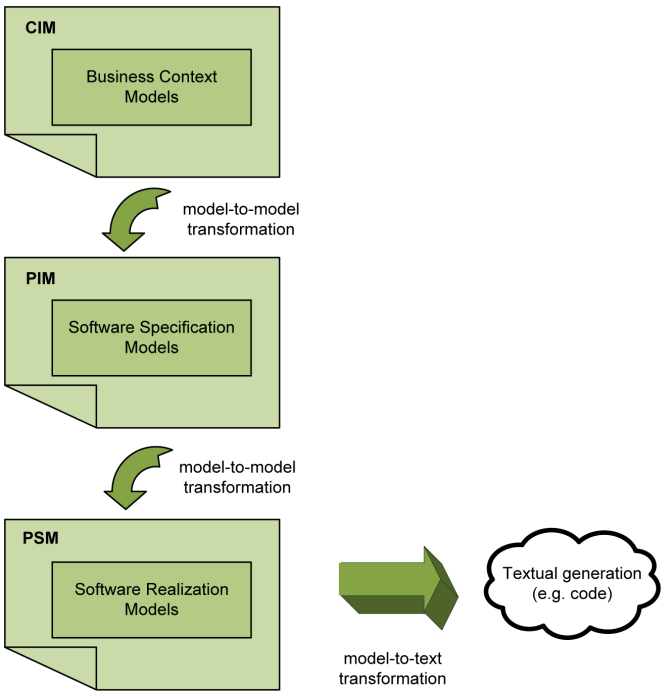


Figure 3.2: *Simplified transformation from CIM to textual generation.*

and for reasons of longevity in the rapid change of the business development, MDA defines three main levels of abstraction. These are:

CIM The Computational Independent Model (CIM) focuses on the environment of the system and hides the structural details regarding the implementation platform. It captures the business context and the business requirements. Usually, the CIM is constructed by a business analyst.

PIM The Platform Independent Model (PIM) describes the system from a platform independent perspective. It captures the abstractions of one or more platform, by hiding the platforms specific data. The platform in this context is the set of technologies and subsystems that provides the needed functionality. A PIM is mostly used in describing the architecture of a system, including its operation and details, but without any specific implementation data.

PSM The Platform Specific Model (PSM) is the representation of the system including the platform specific data. It combines the specification of the PIM with the details of a specific platform. In the MDA context, a PSM is created via a model transformation from a PIM. For instance, a PIM describing the domain model of a system can be transformed to a specific implementation platform like J2EE or .Net.

3.1.1.3 Model Transformations

As depicted in Figure 3.2, we can go from one viewpoint to other by doing transformation. There exists two kind of transformations; (i) model-to-model which defines mapping rules between concepts in source to target model and (ii) model-to-text for textual generations (e.g. code for specific application language). Note that model-to-model transformation is not only used to move from one abstraction level to another, but could also be used between models at the same level, for instance PIM-to-PIM or PSM-to-PSM.

Several model transformation languages have emerged along with Model-Driven Engineering (MDE). Some of them specifically target the definition of model-to-model transformations, like The Atlas Transformation Language (ATL) [43]. Other are targeted for the definition of model-to-text transformations (e.g. code or documentation generators), like MOFScript [23]. Most of the transformation languages are simple, rule-based languages and their modularization constructs do not scale as the transformations get complex or large in size.

3.1.2 Software Factory

Microsoft introduces Software Factories (SF) [27] as a new software development paradigm. SF primarily focuses on Product Line Development, which copes with developing a set of similar but distinct products. In this context SF relies heavily on models and automation, which are basic concerns of MDD.

Jack Greenfield and Keith Short define the methodology Software Factory [27] as followed: “A Software Factory is a Software Product Line that configures extensible tools, processes, and content using a software factory template based in a software factory schema to automate the development and maintenance of variants of an archetypical product by adapting, assembling, and configuring framework based components."

A Software Factory has two central elements, (i) a *Software Factory Schema* and (ii) a *Software Factory Template*. A SF Schema defines, categorizes and summarizes the artifacts and assets required to build a software product line. It can be seen as a recipe listing ingredients, tools and the application process. A SF Template is based on the SF Schema and represents the implementation of the SF Schema that means that all defined assets and artifacts have to be built and made available. The implementation comprises among others developing DSLs. The SF Template can be seen as a bag of groceries containing the ingredients listed in the recipe (SF Schema).

A core principle of the SF approach is to enable a high degree of reuse of existing assets and development of new reusable assets. The development of a specific member of a product family comprises reusing existing assets and developing variable assets for that specific member. The SF approach

uses the concept of *Domain-Specific Modeling* (DSM), which utilizes *Domain-Specific Languages* (DSLs) for modeling.

3.1.2.1 Domain-Specific Modeling (DSM)

The purpose of DSM serves primarily for the creation of models for computation and secondarily for documentation. Models should be able to be processed by tools to generate source code. DSM aims to bridge the gap between the problem domain and the solution domain by the means of abstraction. DSLs raise the level of abstraction by incorporating domain concepts. A DSL uses concepts and terms of the problem domain and provides specific graphical and textual notations to create models, thus it is very close to the problem domain. The more a DSL can be tailored to a specific domain or purpose the more efficient the DSL and the code generation will be.

The architecture of a DSL is similar to UML's architecture, which is also represented by a four-layer hierarchy. The significant difference is that the metamodel layer (corresponding to UML's M2 layer) has to be defined, while UML's metamodel layer is already defined and can be extended by Profiles. The mapping of a DSL model into a programming language is conducted by an appropriate code generator, which must be designed and developed for a given DSL or at least be configurable to tailor its task to a specific domain. A DSL is mapped to a programming language, just as a programming language like Java is mapped into byte code. According to this analogy, the compiler would correspond to the code generator.

3.1.3 Summary

Available MDA tools [1] demonstrate that productivity can be increased by applying the MDA approach. Certainly, the utilized tool plays an important role, but a better productivity can be achieved particularly due to the omission of the implementation phase. A disadvantage in this case is that the learning phase for the MDA tool is due to its complexity and individuality very time-consuming. Nevertheless, the modeling process can be started immediately, since the modeling language (UML) is already provided, apart from the fact that specific UML Profiles are needed and not granted by the MDA tool. As we described earlier in Section 2.6, tools supporting complete UML specification and UML Profiles are still missing, and are maybe the biggest challenge when using the MDA approach. Also, each tool has it's own way of implementing and defining UML Profile, which extend the learning phase further.

The SF solution shows that the SF methodology can increase productivity as well, basically for the same reason as in the case of MDA, the implementation part is omitted. But, before reaching this point, the expense for

[1] http://www.omg.org/mda/products_success.htm/

the SF approach is much higher, because the DSL has to be developed first, which is time-consuming and sophisticated, and requires expert knowledge about the problem and the solution domain. This fact delays the start of the modeling process.

Once a DSL is created, a better efficiency can be achieved, because a DSL comprises domain concepts and thus it is closer to the problem domain. This fact facilitates involving business stake holders into the specification process to avoid misinterpretation and confusion. This is an advantage for DSLs, since the comprehension of UML models require UML experts.

Quality and reliability of a software system can be improved as well in both approaches, particularly due to the reason that the generated code is less incorrect, because of its generation according to a scheme, rules or code-templates. Provided that the model interpreter works properly, the generated code is more reliable than handcrafted code that usually contains bugs, because a developer tends to make mistakes. Furthermore, the generated applications exactly meet their specification in form of models, since they are generated according to them.

Certainly, MDA and SF apply similar methods and techniques for modeling and mapping, but they have distinct objectives. Due to the high expense of developing a DSL and a code generator, the SF approach is only recommendable for developing Product Lines, because this expense has to be compensated somehow. However, once a DSL and the code generator are developed, the costs for generating the product line members are very low due to their similarity. The overall costs of a product line development can then be distributed on the amount of all members, which makes the SF approach productive. Otherwise, for One-Off development, the SF approach would be expensive in terms of time, budget and resources. MDA on the contrary can be used for One-Off Development and Product Line development, since there is no additional expense to compensate. Product Line development with the MDA approach would even increase the regular expected degree of productivity, because the first model could be reused for the other product line members.

Fact is that MDA is a pure MDD approach and focuses on platform independence, while SF is an entire software development methodology and focuses on product line development. UML as the standard modeling language for MDA is a general purpose language, which has to be specialized and constrained with Profiles to be appropriate for MDD. A DSL in contrary is supposed to be developed for a specific domain from beginning, without specializing and constraining afterwards, in this manner DSLs can be very efficient within that domain, but also very useless in other domains. On the whole, both approaches have their strengths and weaknesses; none of them is clearly in advance. Depending on the purpose they are applied for, they demonstrate different strengths and weaknesses. An appropriate problem domain, professional developers, a suitable tool and a precise idea of the

intended products or product family, can guarantee each approach's benefits.

3.2 Service-Oriented Architecture (SOA)

It is hard to find a comprehensive definition of Service-Oriented Architecture (SOA) because up to now, SOA has not been standardized. Therefore, this section introduces the most important and widely accepted concepts and ideas behind SOA. Moreover, this section gives a brief overview how SOA-based systems usually look like in practice.

SOA is not new, it has been around for many years, but it has gained popularity due to the increase of the complexity in today's software systems and [46] introduces SOA as a concept whose time has come. SOA is not something magic, and certainly not a "fix" to all problems in enterprise systems, but a vision and guideline that allows the IT-functionality to be delivered as modular business services in order to achieve specific business benefits.

But what is SOA? If you google this question you will get around 9 million results, and hopefully not that much different answers, but many enough to conclude that there is no common definition. However, according to [46], it is a conceptual, technology independent, business architecture that allows business functionality or application logic to be available through reusable IT services. It is a concept, guideline, pattern, an approach and a philosophy of how IT functionality can be planned, designed and delivered in such a way that it will achieve a specific business goal. Therefore it assures **interoperability**, **reusability** and **integration** across all business processes and technology platforms.

According to the OASIS [2] SOA Reference Model [53], SOA is a paradigm for organizing and utilizing capabilities of different ownership domains. It provides the means of organizing solutions owned by others, which combined with locally "owned", enable a more valuable usage of these solutions. The main concept of SOA is the **service**. It is the centerpiece of SOA and considered also as its primary architectural asset. We will base our description of the service and other relating concepts in the reference model introduced below.

3.2.1 SOA Concepts

The *Reference Model for SOA* is an abstract framework for understanding significant relationships among the concepts of the service domain. In the lack of a standard, the abstract framework aims to provide the essence for SOA, in a vocabulary and a common understanding of its concepts. It provides an abstract model not connected to any various existing or future

[2] Organization for the Advancement of Structured Information Standards

deployment technology for this architecture.

Figure 3.3 shows the concepts of the SOA Reference Model. To reduce the complexity of the figure, we did not show concepts related to the *Execution Context*.

A *Service* is described in the reference model as a mechanism that enables access to a set of capabilities. The capabilities incorporate the business behaviour, while the access to them is provided using prescribed *interfaces*. The service is provided by a *service provider* and invoked by the *service consumer*.

Except the *Service description* that specifies the capabilities offered, constraints and policies, and the information needed to use the service, the rest of the information is hidden to the service consumer.

In the context of service dynamics, there are some other concepts involved in the interaction between services: the *visibility* between exchanging parties and the actual interaction between them. The visibility relies in the ability of providers and consumers to interact with each other, while the *interaction* itself involves performing actions against the service. In many cases the *interaction* takes place through message exchange.

In order to describe the interaction process, the service description references an *information model* and a *behaviour model*. The *information model* of a service describes the information that will be exchange with the service. It includes both the structures(syntax) and meaning(semantics) of the information to be exchange. The *behaviour model* provides the knowledge of the actions invoked against a service, including the process of interacting with the service.

The interaction itself is described in the *execution context*. The execution context of a service interaction is the set of infrastructure elements, processes and entities that form the path between the service provider and consumer.

Other concepts about the service are the *service functionality*, included in the *service description* and *policies & contracts*. The policy represents some constraints or conditions on the usage of the service, while the service contract is the agreement by two or more parties.

After introducing the service and other concepts related to its description or interaction, we will have a look at some of its characteristics.

3.2.2 Service Characteristics

This section introduces the service concept like it is utilized in SOA. A service can encapsulate different amounts of business logic: a single process step, larger parts of a business process, or a complete process (see Figure 3.4).

Service-orientation is the foundation for SOA. The difference between SOA and former distributed computing approaches is the way how components of SOA systems are designed. This section explains the most important

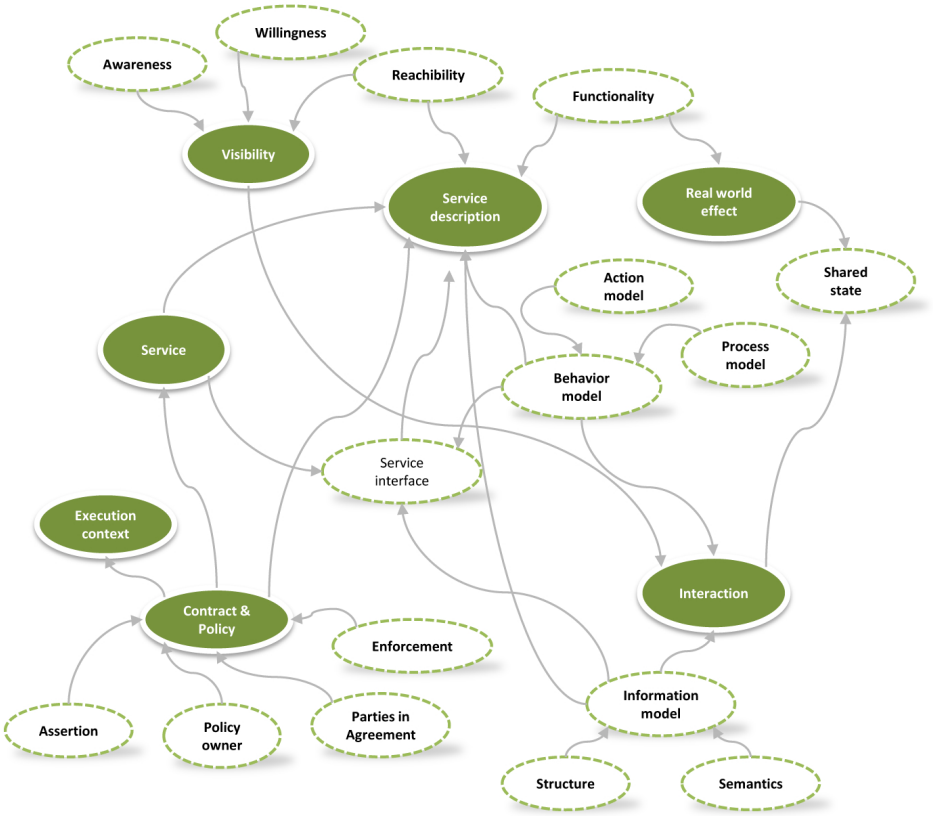


Figure 3.3: *The Concepts of the OASIS Reference Model for SOA.*

service-oriented design concepts for services in the context of SOA. Moreover, we show how SOA systems look like in practice. According to [46], here is an overview of the most important design concepts of services in SOA:

Coarse grained services is one of the most used words in SOA. This characteristic involves the functionality the service encapsulate. The service represents business functions and processes. Other components or services are also contained within functions and processes. The granularity depends on the functionality of the service. For instance too coarse-grained services will become “heavy" and encapsulate a large number of components and transactions, while fine-grained services have a narrow scope and fewer elements.

Contracts: Well defined service contracts involves having a clear specification on the functional and technical details needed to consume a service. It gives the outside world the needed information to use the service, while hiding the technical details.

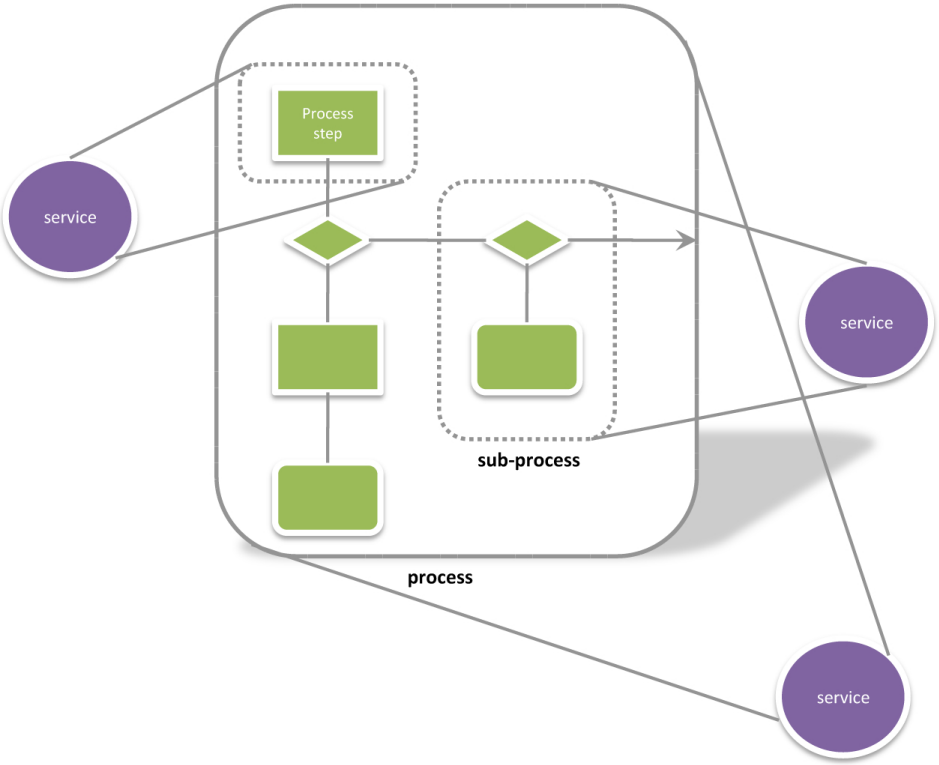


Figure 3.4: *Granularity of Services.*

Loose-coupling: Loosely coupled services is a term that in the context of service design, implies for the service to have a specific implementation. This characteristic describes the service as an entity not bound to the implementation technology. In the need of change this implementation could be replaced without any further changes in the architecture of the system.

Discover-ability: Discoverable in the service aspect means that the service constructs or service description except for being well designed should also be published and visible to the consumer. They should be easily discovered implying that the services are listed in service registries in order to achieve a bigger audience than the one intended in the design.

Compose-ability: Composable is a term use in SOA to characterize a service as a collection of other components or services. The service contains the components that form its functionality, but other services may also be incorporated in it. This leads to an incremental extensibility of services.

A service in SOA is considered to be business aligned. It should en-

capsulate the business requirement, behaviour and other imperatives which are found in business modeling. In order have a long lasting alignment, the service should be durable. The durability of the service do not imply it to be rigid and not flexible, but to hold the relationship to the business it represents.

The service in SOA is reusable and interoperable. These are the main values that SOA promotes, but also the most difficult to achieve. The reusability implies incorporating the proper functions of the services, in the way that it can have a clear definition of its use and multiple consumption partners. The interoperability involves clear definition of the application policies and standards of the services.

3.2.3 SOA Modeling and Implementation

SOA has been associated with a variety of approaches and technologies. Web Services are one of many implementations of SOA, but many people think that this is actually SOA. The lack of common definition of SOA has made the work difficult with respect to Model-Driven Architecture (MDA) approach of building platform independent model. However, one of the recent results on this field is the *Platform-Independent Model for Service-Oriented Architecture* (PIM4SOA) which were developed during the ATHENA project at DFKI[3]. The project did not only build PIM, but also graphical editor in Eclipse environment to support underlying language. But, the problem was that metamodel was large and complex, and the service concept almost didn't exist. Its expressiveness is limited to the design of rather simple scenarios

November 2006, the Object Management Group (OMG) started a standardization process for a platform independent model for services, called UPMS (*UML Profile and Metamodel for Services*) [56]. The main objectives of this new standard for services are (i) to enable interoperability and integration at the model level, (ii) to enable SOAs on existing platforms through OMG's MDA initiative, and (iii) allows flexible platform choices.

Initial submission to the RFP was submitted by IBM, together with some other partners and concentrated most on metamodel. This submission has lately been merged with the submission from Adaptive, EDS and Model-Driven Solutions and are currently under review. The focus now lies on UML Profile since it's more natural to use profiles when extensions are related to UML, and it's easier to apply it to existing modeling tools. By the end of 2008, OMG should vote and adopt specification that covers the requirements from RFP and seems to be most feasible.

A brief overview of this UML Profile and Metamodel for Services (SOA-Pro) [17] is given in next sections and is also the base for further discussions in this thesis.

[3]German Research Center for Artificial Intelligence

3.2.4 UML Profile and Metamodel for Services (SOA-Pro)

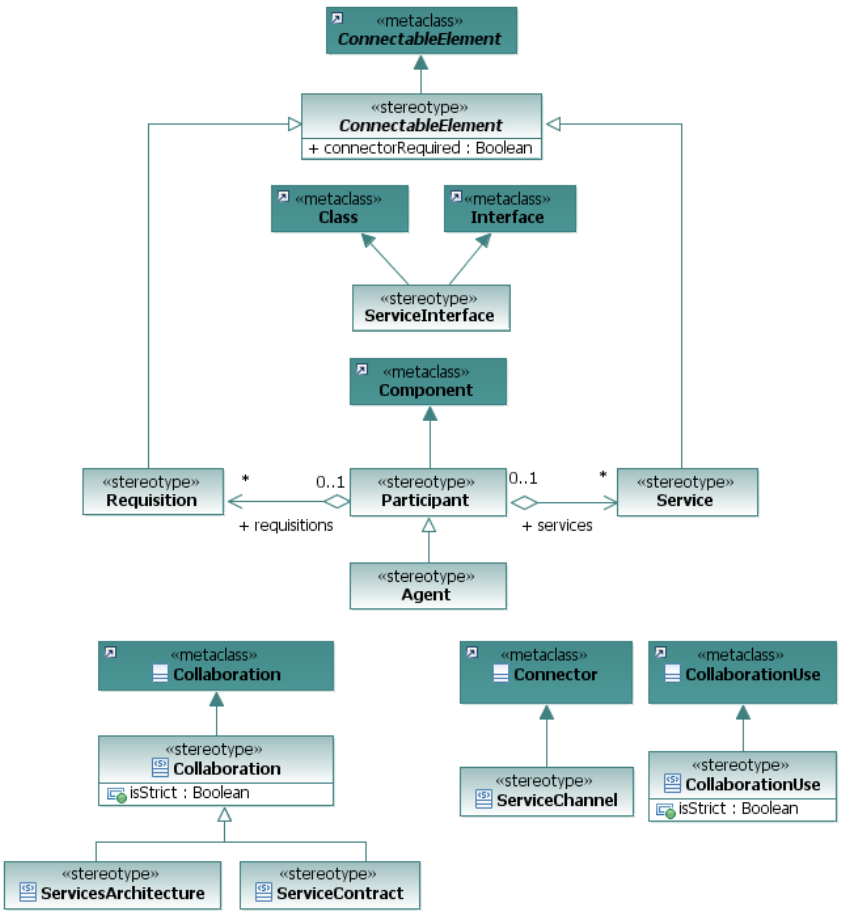


Figure 3.5: *Core concepts of the profile of SOA-Pro, UML Profile and Metamodel for Services.*

In general, the UML Profile and Metamodel for Services (SOA-Pro) is based on the UML 2.0 metamodel L2 and provides minimal extensions to UML, only where absolutely necessary to accomplish the goals and requirements of service modeling. The specification takes advantage of the package merge feature of UML 2.0 to merge the extensions into UML. The profile provides a UML specific version of the metamodel that can be incorporated into standard UML modeling tools. In the following, we discuss the core concepts of SOA-Pro in more detail.

3.2.4.1 Introduction

As already mentioned, SOAs are a way of organizing and understanding organizations, communities and systems to maximize agility, scale and inter-

operability. SOA-Pro claims to be very flexible in order to support activities of service modeling and design, and to fit into an overall Model-Driven Development approach. It supports three different perspectives: (i) from the service consumer requesting the service, (ii) service provider advertising a service to those who are interested and qualified to use it and (iii) from a system design describing how consumers and providers will interact to achieve overall objectives.

Furthermore, it supports both, *bottom-up* and *top-down* architecture designs of services. In the case of the *bottom-up* definition, services are context independent and focus on their own specification which makes them very simple. In contrast, the *top-down* definition enables an organization or community to work more effectively using inter-related sets of services and focus "in the large". Figure 3.5 depicts the core concepts of SOA-Pro that are discussed in the following in more detail.

3.2.4.2 Basic Services

The key concept of a service is a capability offered by one entity or entities to others using well defined "terms and conditions" and interfaces. Those entities may be people, organizations or systems and are called *participants*.

In SOA-Pro, capabilities are provided or required by participants through the interaction points, the UML Ports. To express that participants offer service capabilities, the certain Port is stereotyped as `<<Service>>`. In contrast, if the capabilities are required by the participant, the Port is stereotyped as `<<Requisition>>`. The `<<Service>>` port is the interaction point where consumers of the service use that particular service. The Port has a type, which describes how to use that service that may be either a UML Interface (for very simple services) or a ServiceInterface (see Section 3.2.4.3). In either case, the type of the service port specifies directly or indirectly, everything that is needed to interact with that service—it mainly describes the contract between the providers and consumers of that service.

3.2.4.3 Service Interfaces

The capabilities and needs of a Service or Requisition port are defined by its type, which is a *ServiceInterface*, or in simple cases, a UML 2 interface. In this case, there is no protocol associated with the Service. Consumers simply invoke the operations of the ServiceInterface. A ServiceInterface may also be a Class which may specify various protocols for using the functional capabilities defined by the servicer interface. This provides reusable protocol definitions for different service participants providing or consuming the same Service.

The ServiceInterface stereotype is like an UML Interface, but has the additional feature that it can specify a bi-directional service—where both,

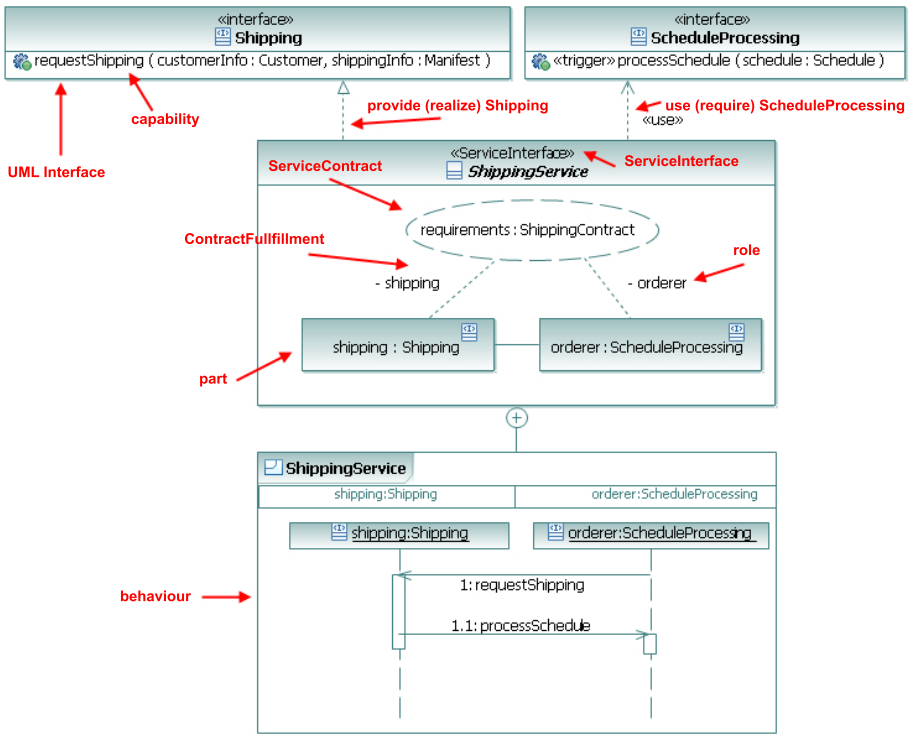


Figure 3.6: *ServiceInterface with UML Interface, CollaborationUse, Part and sequence diagram for describing behaviour.*

the provider and consumer, have responsibilities to send and receive messages and events. The ServiceInterface is defined from the perspective of the service provider, using three primary sections:

Interfaces are standard UML Interfaces that are realized or used by the ServiceInterface. The interface that is realized specifies the provided capabilities, the messages that will be received by the provider (and correspondingly sent by the consumer). The interface used by the ServiceInterface defines the required capabilities, the messages or events that will be received by the consumer (and correspondingly sent by the provider)

ServiceInterface and enclosed parts specify the roles that will be played by the parties involved with the service. The role that is typed by the realized interface will be played by the service provider

Behavior specifies the interactions between the provider and consumer—the contract or protocol of the interaction, without specifying how either party implements their role. Any UML behavior specification can be used, but interaction and activity diagrams are the most common

3.2.4.4 Participants and Service Ports

The *Participant* stereotype in SOA-Pro is specifying the UML Component classifier, and is used to represent software components, organizations, actors or individuals in SOA. The Participants are defined by the role they are playing in the service architecture, and the capabilities they are providing and requiring.

Just as we want to be able to describe what capabilities of the Participant are provided, using the `<<Service>>` port, we also want to express what capabilities are consumed. This is defined by stereotyping the port as a `<<Requisition>>`. The type of a `<<Requisition>>` port is a ServiceInterface or UML Interface as well as it is with a `<<Service>>` port. The `<<Requisition>>` port is the conjugate of a `<<Service>>` port in which it defines the use of a service.

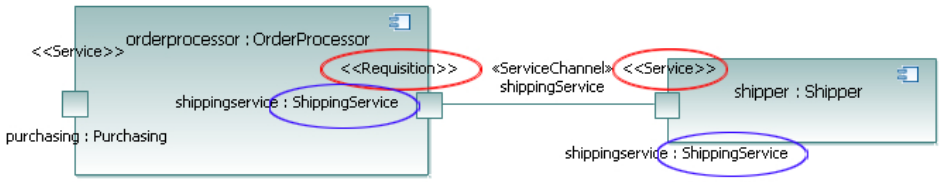


Figure 3.7: *Participants connected to each other through UML Ports with same Port type.*

To illustrate why both stereotypes are needed, we take a closer look at Figure 3.7 that illustrates two Participants, *OrderProcessor* and *Shipper*. The interaction between them is done through their UML Ports. *Shipper* is providing the capabilities described in *ShippingService* (see Figure 3.6) and therefore its port is stereotyped as Service. In contrast, *OrderProcessor* is requiring the capabilities of the *ShippingService*, and its port is stereotyped as Requisition. Note that Participants can only interact with each other through ports with the same type and opposite stereotype of the port.

In Figure 3.6, we described the capabilities of the *ShippingService* only once. Certainly, this ServiceInterface is defined from the perspective of *Shipper*, but with this technique, we are able to specify that it is used by the *OrderProcessor* Participant. Without this technique, we would have to describe the capabilities needed and provided twice, from *Shipper* and *OrderProcessor* perspectives and that would make the models more complex and harder to maintain.

3.2.4.5 Service Contracts

A *ServiceContract* is the specification of the agreement between consumers and providers of a service as to what information, products, assets, value and obligations will flow between them - it specifies the service without regard for realization or implementation.

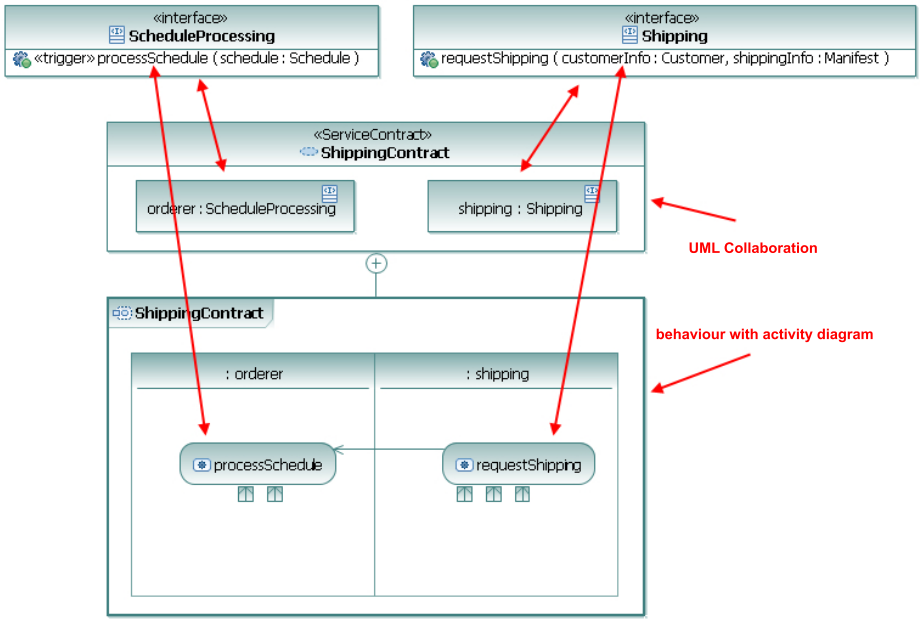


Figure 3.8: *Service Contract, defining the roles and the interfaces.*

As illustrated in Figure 3.8, the basis of the ServiceContract, *Shipping-Contract*, is the UML Collaboration that focus on the interaction involved in providing a service and where the particular roles are defined. The participants fulfills the ServiceContract by the corresponding CollaborationUse, as depicted in Figure 3.6.

3.2.4.6 Service Architecture

A *ServiceArchitecture* (SOA) describes the roles of a set of Participants that provide and use services to achieve some mutual goal or implement a business process. By expressing the use of services, the ServiceArchitecture implies some degree of knowledge of the dependencies of the participants, i.e., the consumption of services to deliver their services. Each service of a ServiceArchitecture is represented by the use of a ServiceContract bound to participant roles.

3.2.5 Summary

In this section we described main concepts of SOA, including the service as the centerpiece, and concepts of service interaction or service description. Later in this section we introduced the main characteristics of a service and concluded with the relation of SOA to models and implementation technologies.

SOA promises the integration of business processes inside an organisation and across organisations. Today's SOA systems are usually implemented with Web Services. There are some problems that are hard to solve with the Web Services technology. For example, the dynamic composition of Web Services is an unsolved issue. Due to the fact that SOA is basically technology independent, one can rise the question if other technologies should be utilized instead or in addition to Web Services to solve the open problems. Agent technology offers several benefits over Web Services and could be used to overcome some of the most important problems of today's SOA systems.

3.3 Agents and Multi-Agent Systems

This section introduce the basics about *agents* and *multi-agent systems* (MASs). We begin with Section 3.3.1 by answering the first obvious question, namely "*What* is an agent?". We answer this question by first introducing four relevant definitions. Then we will discuss properties that characterize an intelligent agent and contrast agents with objects and web services in Sections 3.3.1.3 and 3.3.1.4.

The usual second question is "*Why* should I bother with agents?". In Section 3.3.2 we answer this question by arguing that agents are a natural progression from objects that provide a better abstraction and improved encapsulation. The remaining subsections of this section are dedicated to answering the third question "*How* do I develop agents and agent systems?".

3.3.1 What is an IntelligentAgent?

Let us first note that we are talking about *software agents*. Whenever we say "agent", we really mean "software agent". The typical dictionary definition of agent as

> *"An entity having the authority to act on behalf of another."*

(e.g. a real estate agent) is *not* what we mean. Although, some software agents may act as agents in this sense as well. For example, a software assistant that buys products or services on behalf of its user.

Software Agents are a part of *Distributed Artificial Intelligence* (DAI), which is a sub-field of *Artificial Intelligence* (see [16, page 121-164]) research dedicated to the development of distributed solutions for complex problems

regarded as requiring intelligence. Since the mid of the 1990s, agent technology was successfully applied in many practical applications (e.g. Whitestein Technologies [69]). The focus and use of agents in this thesis lies on the ability of agents to collaborate in and interact with distributed computing environments, such as *SOA*. Agent technology promises the ability of writing software for distributed computing environments in an intuitive and elegant way.

As is to be expected from a fairly young area of research, there is not yet a universal definition of what an agent is. Following we present different definitions, however the Wooldridge and Jennings definition, Definition 4, is increasingly adopted and is the one that we rely on in the rest of this thesis.

Definition 1 *"Agents are* **active**, **persistent** *(software) components that* **perceive**, **reason**, **act**, *and* **communicate**.*"*

— Huhns and Singh, 1997 [68, page 29]

Definition 2 *"An agent is an entity whose state is viewed as consisting of* **mental** *components such as* **beliefs** , **capabilities**, **choices**, *and* **commitments***"*

— Bradshaw, 1997 [8, page 272]

Definition 3 *"For each possible* **percept sequence**, *a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has."*

— Russel and Norvig, 2003 [35, page 34]

Definition 4 *"An* **agent** *is a computer system that is* **situated** *in some* **environment**, *and that is capable of* **autonomous action** *in this environment in order to meet its design objectives."*

— *Wooldridge, 2002 [47] (adopted from Wooldridge and Jennings, 1995 [49, page 25])*

According to the Definition 4, there are two distinguish notions of agency in the relevant literature: the *weaker* and the *stronger* notion, or also referred as *mentalistic* notion.

Attributes describing the *weak* notions is called *weak* because it does not take much to fulfill them and are the general way of characterizing the term agent to denote a hardware or software-based computer system. For example, a simple UNIX service that process these attributes could be called agent. Short explanation of the *weak* properties that are most important in agent applications follows.

Situated: Agents are situated in an *dynamic, unpredictable* and *unreliable* environments. Environments are changing rapidly and agents can't assume that the environment will remain static while it is trying to achieve a goal. This means that it's impossible for the agent to predict what will happen in the future states of the environment. The environments are *unreliable* in that the action that an agent can perform may fail for reasons that are beyond an agent's control.

Reactive: To deal with the situations in such environments as described in previous property, agents must respond to significant changes in its environment, in a timely manner. Agents percept their environment, process their perceptions, and choose an appropriate actions that are performed by effecting the environment.

Proactive: Agents *persistently* pursues goals over time. Because agents are *persistent*, they are more robust and can continue to attempt to achieve a goal despite failed attempts. They are also able to exhibit goal-directed behaviour by taking the initiative, without any external invocation.

Robust: Because of challenging environments agents are situated in, failure of actions and more generally plans to reach the goals is a possibility. In that case, it is important that agent are able to recover from such failures.

Flexible: To achieve this robustness, agents are *flexible* in that they have a range of ways of achieving a given goal should a plan fail or environment be changed.

Autonomous: Agents are *autonomous* - they have control both over their *internal state* and over their *own behaviour*. There is no need for external control by humans or other agents. Autonomy is a defining agent characteristic and a consequence of the characteristics previously listed.

Social: In order to reach the goals, agent can interact with other agents and in that way be *social*. This interaction is often on a higher level. Instead of just saying that agents exchange messages, agent interaction can be framed in terms of *performatives* [47, chap. 8] such as *inform*, *request*, *agree* and so on. Agent interaction are aligned to human interaction types, such as *negotiation*, *coordination*, *cooperation* and *teamwork*. We discuss *Multi-Agent Systems* further in Section 3.3.1.2.

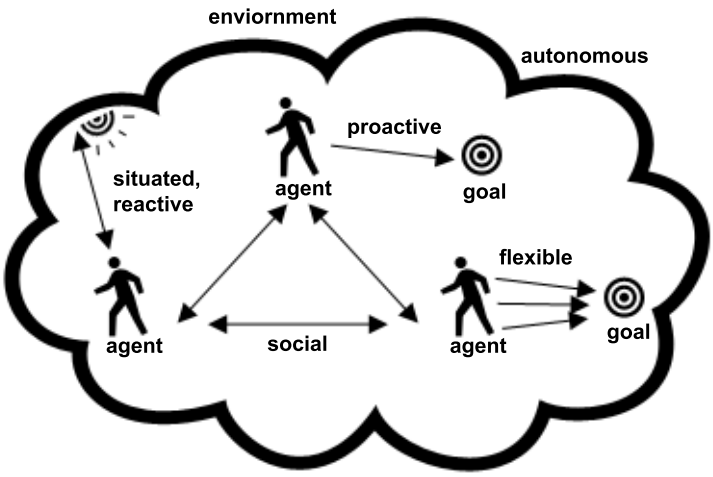


Figure 3.9: *Weak notion of intelligent agents.*

The second notion of agents are the *stronger* notion. Such notion reflects those used in human cognition or communication, such as agents having *beliefs* about the world or certain *desires* or *aims*. Or agents performing *intended* actions to progress towards a goal. This notion might sound somewhat human-like, but it is representing programming abstractions intended to facilitate a more intelligent approach to decision making. According to [16] and [65], we have following explanation of these three *mental* capabilities:

Believe: Beliefs represent the informational state of the agent - in other words it's beliefs about the world (including itself and other agents).

Beliefs can also include inference rules, allowing forward chaining to lead to new beliefs. Typically, this information will be stored in a database (sometimes called a belief base), although that is an implementation decision. Using the term belief rather than knowledge, recognises that what an agent believes may not necessarily be true (and in fact may change in the future).

Desire: Desires (or goals) represent the motivational state of the agent. They represent objectives or situations that the agent would like to accomplish or bring about. Examples of desires might be: find the best price, go to the party or become rich. Usage of the term goals adds the further restriction that the set of goals must be consistent. For example, one should not have concurrent goals to go to a party and to stay at home, even though they could both be desirable.

Intention: In order to reach a certain goal, a BDI agent has to plan its actions. Planning is based on the agent's beliefs. The computed action sequences are the agents intentions.

Jennings and Wooldrige [49] also discussed further properties one could consider useful in the design process of agent-based systems. These properties are not directly linked with the notion of weaker or stronger agency. They are rather orthogonal to the properties discussed so far and may or may not considered to be useful concepts in the context of a specific application.

Because concepts like *goals*, *knowledge*, and *ontologies* are quite naturally encountered in business context as well as in Service-Oriented Architectures, it is obvious that it is natural to adopt the strong notion of agency in this context.

Rationality: Does the agent make its decision according to a rational decision criterion (e.g., utility maximization)?

Truthful: Does the agent communicate in truthful manner with its outside environment?

Benevolent: Is the agent per se benevolent, i.e. it will never intentionally do malicious actions?

Mobile: Is the agent able to migrate from one platform (computer) to another one?

Emotional: Does the agent (pretend) to exhibit emotional states?

Agents receives inputs related to the state of their environment through *sensors* and they act on the environment through *actuators*, like Figure 3.10

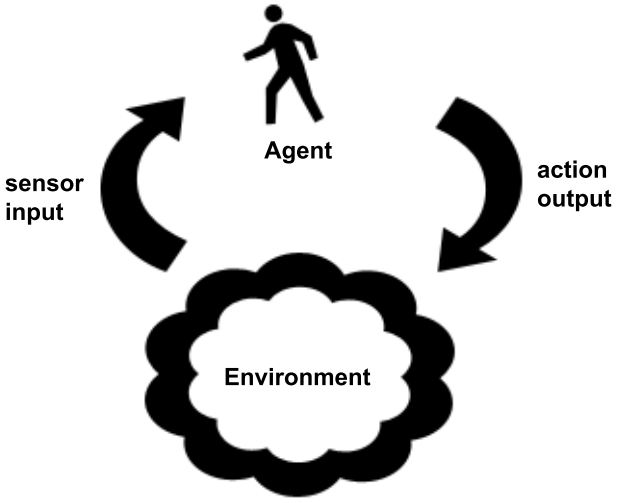


Figure 3.10: *Interaction between agent and its environment through sensor input and action output [16].*

Weak Notion	Stronger Notion	Other properties
Autonomy	Knowledge/Believes	Rational
Social Ability	Intentions	Truthful
Reactivity	Desires/Goals	Benevolent
Pro-Activeness	Obligations	Mobile
Flexible	Capabilities	Emotional

Table 3.1: *Properties of different notions of agency.*

illustrates. According to [35], an agent's *environment* can be *fully* or *partially observable*, *deterministic, stochastic, single-agent, multi-agent*, etc. The environment is usually provided by an agent execution platform.

3.3.1.1 Agent Architectures

In previous section we introduced two notions of agents, which are commonly used. An agent architecture is a blueprint for software agents and intelligent control systems, depicting the arrangement of components. These architectures are often referred to as *cognitive architectures*. Here we will shortly introduce two of many architectures that have properties of the notions mentioned earlier.

Reactive agent architectures is one type of agent architectures that have properties of the *weak* notion of agency. Agents choose the next action to

perform, only based on the current perceptions. They do not remember previous perceptions and states, and are also not able to reason. Figure 3.11 shows the basic architecture and as we can see, *condition-action rules* defines a direct mapping from *perceptions* to an *action*. It is obvious that reactive agents are limited because they can not remember previous states. One of the most successful reactive agent architecture implementations is the *subsumption architecture* which is a reactive robot architecture heavily associated with behaviour-based robotics and was introduced by Rodney Brooks and colleagues in 1986 [11].

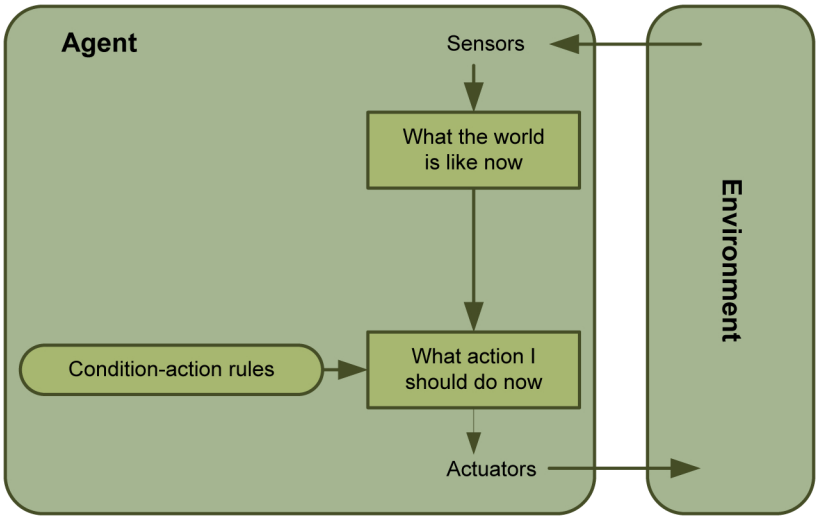


Figure 3.11: *Basic reactive agent architecture (from [35, page 47]).*

Believe-Desire-Intention Architectures is the second architecture type and introduce a more complex design and composition of agents. BDI architectures assume agents to possess *mental capabilities* and are based on the properties from *mental* or the *stronger* notion of agency as described earlier. There exist formal frameworks that specify the semantics of believes, desires, and intentions (e.g. [65]). Moreover, there also exist two important multi-agent frameworks that are based on the BDI architecture: Jack [28] and Jadex [15]. Both multi-agent frameworks have been utilized in many real world scenarios.

3.3.1.2 Multi-Agent Systems

The various agent technologies existing today can be classified as being either *single-agent* or *multi-agent* system. In *single-agent* systems, an agent may interact with the user or with local or remote system resource. In contrast, a *Multi-Agent System* (MAS) is a system designed and implemented as several interacting agents. Hence, *MAS* can communicate either with hu-

man and system resource or with other agent. Consistent with the degree of cooperation exhibited by the individual agent, *MAS* is classified into two types:

Competitive agents: Each agent goal is to optimise its own interest, while attempting to reach agreement with other agents. Example of the system that using competitive agent are:

BargainFinder [5] The user give a specific product that she/he is looking for. The system queries a fixed set of web merchant sites for its price and present the user with a ranked set of price quotes.

Jango It is an advanced version of BargainFinder which makes product

Kasbah [2] Kasbah is a ads service on the WWW that incorporates interface agents. A web site represent a 'market-place' where Kasbah agents, acting on behalf of their owners, can filter through the ads and find those that their users might be interested in. The agent negotiates to buy and sell items.

MAGMA [39] Magma introduces an open marketplace involving agents buying/selling physical goods, investments and forming competitive/cooperative alliances. Magma has "offer board" that contains offers from buyer and sellers. Seller agent posts the highest price of offered product to the offer board through its Personal Assistants (PA). The buyers inform the PA to retrieve a number of best offers of a specific product that they interested in. The buyer PA will display a set of offers. The buyer will select which offer to follow up. Finally, the buyer PA will then post an accept offer to the relevant seller agent.

Cooperative agents: Agents share their knowledge and beliefs to try to maximise the benefit of the community as a whole. Example of the system that using competitive agent are:

Air Traffic Control Chu and Carrol [37; 44] have developed a cooperative negotiation model among the agents (pilot, air traffic control). They cooperate to develop the best plan for flight and handle the entire situation, which affect the flight.

We can classify multi-agent production scheduling in this thesis as cooperative multi-agents, since the agents cooperate to optimise production and execution tasks in time and cost constraint. Communication is essential in a *MAS* to enable social behaviour among the agents. Agents can coordinate their actions to reach common goals and to prevent deadlocks. Collaborating agents can solve more complex tasks than a single agent could solve. This property of *MAS*s is called *emergent functionality*. In Section 3.4 we will introduce two commonly used agent-related interaction protocols in *MAS*.

3.3.1.3 Agents and Objects

In previous sections we have covered the definitions and characteristics of intelligent software agents. The reader might be confused about just how different - or the same are objects and agents. This question has been answered by many authors. We base our answers on this topic according to [62; 34; 26], since they summarize and collect these differences. Two key areas that can differentiate the agent-based approach from traditional Object Oriented are *autonomy* and *interaction* and we will try to cover those in this section. However, there are other ways in which agents may seem to differ from objects (see "Philosophical Differences" [34, page 47]), but we will not include them in this thesis as they are not so relevant for our case study.

Autonomy : In the object-oriented paradigm, the system is comprised of several, possibly distributed, objects interacting with each other through predefined interfaces. The interface describes the object's services which can be invoked by other objects. In other words, an object is *passive* system component which is able to respond to predefined requests and react accordingly. An object system is usually governed by a central process/object responsible for the coordination between objects and the flow of control between them towards the achievement of the desired result. As we have seen so far, agents are being autonomous and therefore agent systems tends to be decentralized.

Another important difference between agents and objects, when we are talking about autonomy, is that objects uses the notion of encapsulation to give objects control over their own internal states. This is done by declaring the access of the methods to be private or public. In this way, objects *are* bringing autonomy, but only over its state. It does not exhibit autonomy over its behaviour and the reason for this is that once the method has been declared as public at the design time, the object has no control over whether that method shall or will be executed in runtime or not. In contrast to agents, objects are not able to this kind of decisions in runtime. In *MAS*, where an agent *requests* an action to be performed by another agent rather than *invoking* this action, request could be *accepted* or *refused* by the serving agent according to various factors such as domain state or the identity of the client.

Interaction : Agents are social entities, which in both notions, strong and weak, focus on the ability to communicate with the environment and other entities (agents). This ability can be also expressed in degrees as illustrated in Figure 3.12. On the left end, we have method invocation that is the usual object oriented way of interacting with other objects. This could be classified as the basic or simple way of interaction. While we are moving to the right end, we are getting more complex degree

of interaction. For example, food-gathering ants don't invoke methods on each other to give instruction for each action one ant should do. Instead, they are listening directly to the physical inputs from the environment, and acting out from this perception. Finally in MAS, agents can collaborate, in parallel, with multiple other agents and act as a society.

An agent message could consist of a character string whose form can vary yet obeys a formal syntax, while the conventional Object Oriented method must contain parameters whose number and sequence are fixed. Agent Communication Language (ACL) is expressive enough to cover all desired situations, including method invocation of objects and expressing communications among multiple agents.

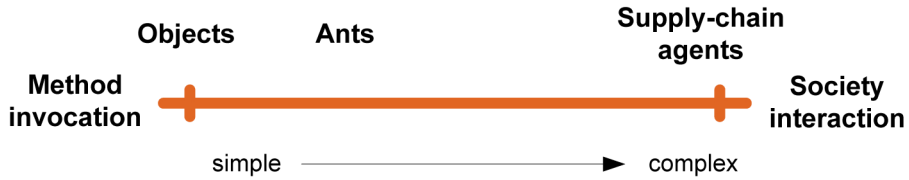


Figure 3.12: *Degrees of interaction, from [34, page 45].*

Conclusion is that agent could appear and have the same capabilities as ordinary object, but it is distinguished from objects since it is autonomous and can interact with the environment and other agents. It is not only contribution to modeling *information* in the enterprises as object often are used for, but it also extends this capability to model *behaviour*. This fact means that the behaviour of agents can be modified dynamically, due to learning of influence of other agents or the environment. Moreover, agents can *dynamically* cooperate to solve problems. In fact, viewpoint in [34] from Odell, considers agents to be an *evolutionary step* forward from objects.

In next section we will relate agents and services that is commonly used in the distributed systems.

3.3.1.4 Agents and Web Services

According to [64], fundamental differences between web services and agents can be pointed out by discussing some of the properties from notions of agency we already have discussed in previous sections.

Flexibility and Robustness The approaches used to engineer agents and web services are fundamentally different. Typically, when designing web services, the work flow to reach a goal is well defined at the design time and any changes in the environment under runtime could cause

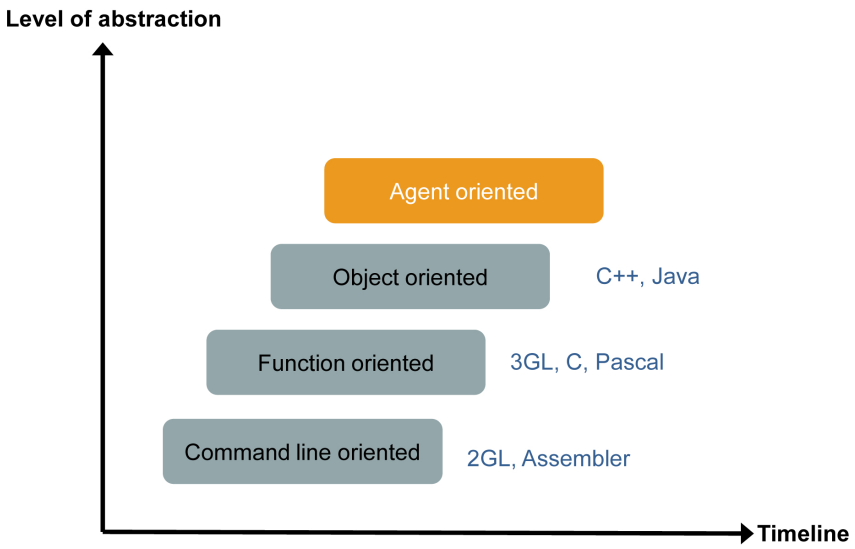


Figure 3.13: *The evolution of programming according to [34].*

failure in the work flow, and the goal will not be reached. Since agents are reactive, flexible social and interacts with the environment, they would adopt to changing contexts and environments and either choose alternative plan or delegate the task to other agents.

Proactiveness Since agents are communicative and social, they respond to both, messages from other agents and changes in the environment and this could trigger their intention to achieve some goal, resulting in proactive behaviour as necessary. Web services are typically just reactive because it has to be triggered by some message, to do some actions.

Goal orientation Typically web services exist to provide access to some resources or facilitate third-party trading between companies. Web services are more task oriented, and will generally perform the task immediately in contrast to agents, which is goal oriented. When agents receives one request, in order to maximize some utility through rational behaviour, it can refuse to execute the task if it doesn't give any gain to reach the overall goal.

Autonomy In the same way as objects, web services needs to be invoked or triggered by some external parties to do some behaviour. As we have pointed several times in this section, agent can evolve its own behaviours without direction from its owner or user.

While in [64] we can read about the differences between agents and web services, in [48] we additionally see how these differences can become blurred

by using agents as an layer over web services as illustrated in Figure 3.14.

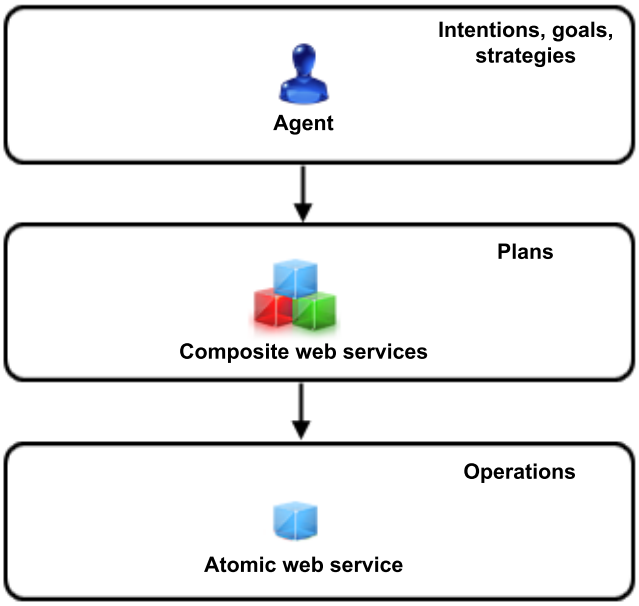


Figure 3.14: *Layered view of agent-ws interactions.*

A key proposal of *Web Service Architecture* (WSA) is that simple, atomic, services can be composed together, in a work flow, to form complex composite behaviours [29]. Research in [48] suggests that BDI Agent performs the planning on behalf of a user to meet some set of goals. This means that agents primarily are responsible for mediating between users' goals, and the available strategies and plans. Agents *invoke*, or *design*, atomic or composite web services as necessary.

3.3.2 Why are Agents Useful?

Having described *what* agents are, we now turn over to the question of *why* agent technology is useful. It is important to realize that, like other software technologies such as objects and web services, agents are not magic. They are simply an approach to structuring and developing software that offers certain benefits, and that is very well suited to certain types of applications.

Agents are reducing coupling since they are autonomous. This can lead to software systems that are more **modular**, **decentralized** and **changeable**. This has led to the application of agents as an *architectural glue* in a range of software applications. In this usage, agents are often used to "wrap" legacy systems.

In addition to reduce the coupling, agents are well suited in applica-

tions where environment is challenging and changing rapidly. The failure is possibility and it's important that recovery is done autonomously.

Properties of being *proactive* and *reactive* is making agents behaviour and characteristics very human-like. This has led to a number of applications which software agents are used as *substitutes* for humans in certain limited domains. The recent computer game *Black and White* [25] used agents, specifically based on the *Belief-Desire-Intention* (BDI) model.

Another application areas where software agents can provide benefits include Intelligent Assistants, Electronic Commerce, Manufacturing and Business process modeling [62].

3.3.3 Agent-Oriented Software Engineering

Software development belongs to the most complex enterprises. Software engineering provides *tools*, *methodologies*, and *principles* for developing software systems. *Agent-Oriented Software Engineering* (AOSE) introduces the agent concept to traditional software engineering approaches [52].

Figure 3.15: *The metamodel reflecting the* **agent** *aspect of the Pim4Agents metamodel.*

The main goal of AOSE is to adapt traditional software engineering approaches to agent technology. Similar to traditional software systems, MASs can be specified with metamodels (see Section 3.1.1). Agent metamodels can be utilized for the analysis and development of MASs. There exist several metamodels for MASs that were developed in the recent years. Some examples are *Gaia* [50], *Prometheus* [62], *Tropos* [10] and *Platform Independent Metamodel for Agents* (Pim4Agents) [12]. The three first mentioned examples are not just metamodels, they are complete methodologies for AOSE. Since our work in this thesis has been in cooperation with *German Research Center for Artificial Intelligence* (DFKI) we want to give an short overview of *Pim4Agents* metamodel (see Figure 3.15).

Concept	Notation	Relation
Agent		performs → Role has ---- Capability behavior → Behavior instance ---- Instance
Capability	Capability	none
Instance		none
Plan		none
DomainRole	DomainRole	capability → Capability

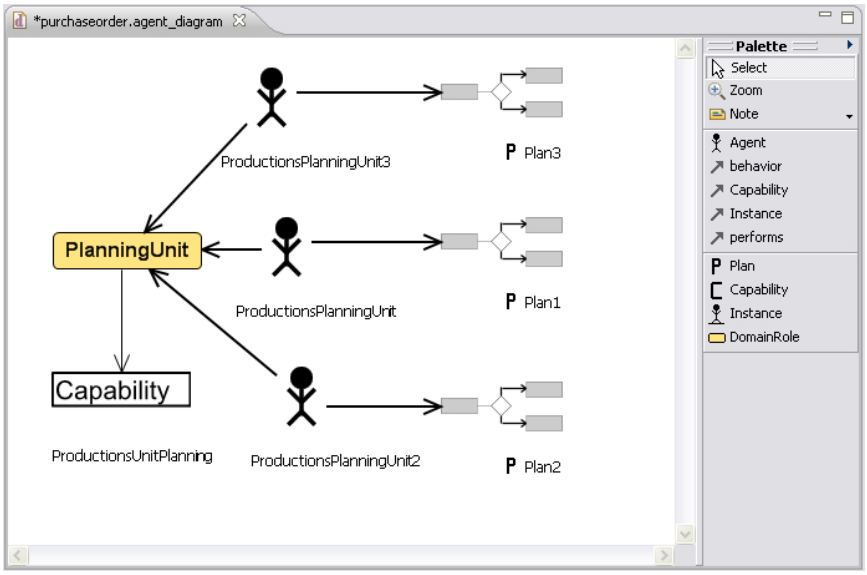


Figure 3.16: *Concepts, notation and instance of the* **agent** *diagram.*

The *Pim4Agents* metamodel specifies the concepts and relations of a typical MAS. One of the central concepts is the *Agent*,the autonomous entity capable of acting in the system. An *Agent* is member of an *Organization* and can perform some *Roles*. The *Role* concept is used to specify responsibilities and permissions. Furthermore, an *Agent* has access to a set of *Resources* which may include information or ontologies from its surrounding Environment. *Behaviours* defines how particular tasks are achieved and at least *Agent* may have certain *Capabilities* that group a particular type of Behaviours.

In addition to the *abstract syntax* of this language, which is expressed in *Eclipse Modeling Framework* (EMF), DFKI has also implemented the *concrete syntax* [67] with help of *Eclipse Graphical Framework* (GMF). Figure 3.16 shows the concepts, notation and instance of the agent diagram in the concrete notation of Pim4Agents. In Chapter 5 we will come back to this language and relate it to our work in this thesis.

The *PIM4Agent* metamodel is platform-independent. Multi-Agent Systems that is modeled with an agent metamodel like *Pim4Agents* can be executed with an interpreter or can be transformed to a specific multi-agent platform like *Jack* [42]JACK [28] and *JadeJava Agent DEvelopment Framework*. AOSE is an important step towards the utilization of agent technology in real world business scenarios. This section gave a rough overview of AOSE. The next section summarizes the agent technology part.

3.3.4 Summary

In this section, we gave a brief overview of agent technology. In Section 3.3.1, we introduced the agent concept and followed with Section 3.3.1.1 where we described different notion of agency. The basic terminology of MASs were introduced in Section 3.3.1.2. Moreover, we presented in Section 3.3.1.3 and 3.3.1.4 the differences between agents compared with web services and objects. After discussing *what* agents are, we continued with *why* they are useful in Section 3.3.2. Finally, we gave an overview of AOSE in Section 3.3.3. We explained the importance of metamodels in AOSE and gave the *Pim4Agents* metamodel as an example.

In following section we will take an further look at the interaction between agents and explain two of the most used interaction protocols in agent systems.

3.4 Optimization with Multi-Agent Systems

Now that we know more about what agents are and how they can be useful in software development, we will in following sections explain how they can be used to optimize supply-chain process in steel production, where the interaction is described through two well-known agent interaction protocols.

3.4.1 Use of Multi-Agent Systems in Transportation Scheduling

Transportation scheduling problems are common real world problems. One of the examples is that in a shipping company and courier service, there are many orders that have to be sent to specific destination address and time. All of these orders should be assigned to the appropriate vehicle such as truck, bicycle, etc. Every vehicle has to be assigned to the appropriate route, so the truck will not spend off the road or travelling empty. Dispatch officers often face some problem when:

- during the execution of the order, a customer comes with a new order
- the vehicle has broken down
- the traffic situation changes

Therefore, the dispatch officers needs to rescheduled the existing plan.

In the last years, transportation problems have been analysed in details within the area of artificial intelligence. In the field of *Multi-Agent Systems (MASs)* and *Distributed Artificial Intelligence (DAI)* there are several recent papers that deal with transportation scenarios. In addition, the use of *MAS*s is an approach that matters more and more in our days. In several later papers the sense of such kind of systems for transportation scenarios came clear. So in [32] and [38] systems are introduced, which are *MAS*s that deal with routing problems. In the year 2006 the [3] workshop, gave several approaches, how agents in a MAS can be used to deal with traffic and transportation problems. The multi-agents can handle the situation where the constraints change, such as the arrival of new order or the changing traffic situation.

For the optimization part of the use case in this thesis, we will use the approach from [32], where the combination of protocols described in sections 3.4.2 and 3.4.3, will be used to optimize the supply chain of steel production.

3.4.2 The Contract Net Protocol

The *Contract Net Protocol (CNP)* [24] is a negotiation protocol proposed by Smith and David in 1980, to specify problem-solving communication and control of nodes in a distributed problem solver.

The *FIPA Contract Net Protocol (CNP)* [21] is a standardised communication protocol done by *Foundation for Intelligent Physical Agents (FIPA)* [19], and is a minor modification of the original, it adds rejection and confirmation communicative acts. One agent, who is the *Initiator*, has some task he wants the other agents (*Participants*) to perform, and further wishes to optimise a function that characterizes this task. In some domains this characteristic is expressed as the price, but it could also be regarding to the execution time, fair distribution of task, etc.

To understand each other, they are using *FIPA acts* defined in the standard *FIPA Communicative Act Library (CAL)* [20]. *FIPA CAL* defines all feasible performatives which are part of the *Agent Communication Language (ACL)* [18].

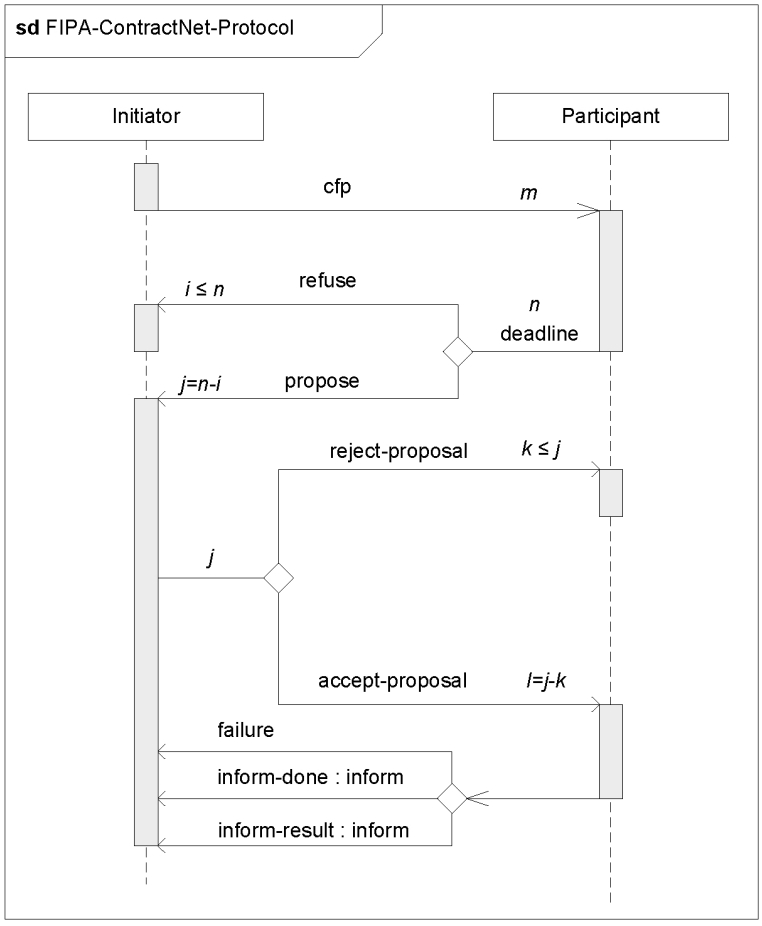


Figure 3.17: *FIPA Contract Net Protocol notation in Agent UML (AUML) [7].*

3.4.2.1 Explanation of the Protocol Flow

The Figure 3.17 illustrates the steps that are performed during a *CNP* run. The protocol is initiated by the *Initiator* which starts with sending m call for proposals (*cfp*) act, which contains a task and conditions, to several other agents (Participants). The potential contractors (Participants) are able to generate n responses where j are to perform the task (*propose*) while i=n-j are to refuse (*refuse*) the task. Attached to the proposal response, agent can for example include the preconditions that are necessary, in order to perform the task, or the costs that would be caused by the task.

Once the deadline passes, the *Initiator* can refuse or accept each of the j proposals separately. The l agents of the selected proposal(s) will be sent an *accept-proposal* act and the remaining k agents will receive a *reject-proposal* act. According to the outcome of the task processing, each *Participant* informs the *Initiator* with one of the following messages; (i) *inform-done* or (ii) explanatory version in the form of an *inform-result*, or (iii) *failure* if the Participant fails to complete the task.

3.4.3 The Simulated Trading Protocol

The *Simulated Trading Protocol (STP)* [1] is a general improvement heuristic for solving vehicle routing problems and was proposed by Bachem, Hochstättler and Malich in 1992. Starting from the initial tour plan it tries to improve the current solution by doing some complex customer interchanges.

In the following three subsections we will give briefly, step for step, description of this algorithm. Basically because we used much resources to get into the details by ourself. One of the main reasons is that the official report refers mostly to the mathematically proofs, and most of the readers will probably find this hard to read and time consuming. However, implementation and use of the protocol in the Chapter 4 and 5, will only focus on message exchange between participants, details about algorithm in Section 3.4.3.3 will be left out.

3.4.3.1 Explanation of the Protocol Flow

The *(STP)* is, like the *CNP*, also a protocol that relies on communication between several entities.

The basic problem of vehicle routing problems is given as follows: a set of n customers with demands di has to be served from a depot using t vehicles of capacity Q. The objective is to minimize the total distance covered by the vehicles or some other measure such as *cost, time, etc.* While traditionally heuristic is only used for local improvement within one of the available tours, *Simulated Trading* introduce a global improvement, which is changing customers between tours.

The algorithm can roughly be described as follows:

Algorithm 1
$T = (T_1,..,T_t)$ is a feasible tour plan
repeat
    Sell-And-Buy phase
    Trading-Matching-Search phase
    If *Trading-Matching M is found* **then**
        Update tour plan T according to M
until *timelimit is reached*

3.4.3.2 Sell-And-Buy Phase

In the *Sell-And-Buy* phase each tour is checked for "good" trading possibilities depending on its previous actions. This defines the trading graph which is searched for a maximum weighted trading matching in the next phase (the

definition and use of the trading graph will be given later in the Section 3.4.3.3). Through one example, we will now discuss these routines in more detail.

Consider the following example of below:

- *Vehicle 1* would like to give up order A since it will give him an extra hour of driving time
- *Vehicle 2* can take order A since position of order A is on the way of his tour. But his vehicle is full and he needs 20 minutes additional time for order B. If someone takes order B, he will be able to insert A in his vehicle with only 10 minutes of extra time.
- *Vehicle 3* is qualified to take order B but if he would insert order B in his tour then he would violate the time window of order C.
- *Vehicle 4* will have an extra hours of driving time if he would take order C. But if another vehicle would be able to take order D then it would be possible for him to take order C.
- *Vehicle 1* is able to take order C by 15 minutes cost.

We can use example above to demonstrate how this customer exchange could be optimized with use of *STP*. As shown in figure 3.18, *depot* starts with creating the initial tour plan for each *vehicle*.

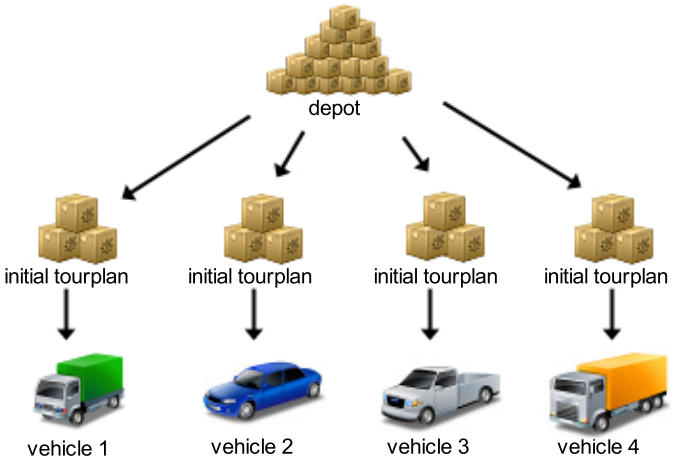


Figure 3.18: *Step 1: Depot creates initial routing plan for each vehicle.*

Since these tour plans are not optimized, the next step is to send a *selling list* to the *vehicles*, containing all orders that have been sold by these, in earlier rounds (or also called *decision level*, more about this in next section).

In the first round, this list is empty and only starts the protocol. After the list with orders has been received, each vehicle decides randomly one, and only one, of the following actions:

(i) **sell**: If an order should be sold, the *vehicle* calculates for each of its orders that could be sold, the *gains* that would be caused by *not* performing the order. The obtained values are the basis for selecting an order and to sell it. Since the *order* to be sold is selected randomly, each *order* has a *probability* to be chosen. This *probability*, depends on the calculated *gains*: the more costs could be saved by selling an order, the higher is the probability for this order to be sold.

(ii) **buy**: If a buying should be performed, the vehicle calculates for each of the *orders* in the *selling* that it could buy, the additional costs this order would cause to its current plan. Depending on these values, one of the orders is randomly chosen to be bought in this round. In this case, the probability for an *order* to be chosen is lower, if it causes more costs when it is inserted into the current plan.

(iii) **wait**: If the vehicle don't want to take any actions in current round, and wants to wait for updated selling list, he answers with *wait*.

After the order to be bought or to be sold is chosen, the *vehicles* informs the *depot* about the result. How these results are used by the *depot* is described in Subsection 3.4.3.3. So, at the end of the round, there are two possible situations:

(i) the *vehicle* has a *new plan*, in case it was able to *buy* or *sell* order

(ii) the *vehicle* sticks to its *old plan*, since it could not *buy* or *sell*, for all buyable orders did not fit into its current plan, or there was no order it could sell

Nonetheless, with the obtained plan, a next round is performed. This *Buy-And-Sell phase* is done iteratively, until a *threshold*, for the number of rounds is reached (normally not more then 10 [32]).

At this point there is a decision for each and for each *vehicle*. All these decisions can be taken into account to find a new solution for the *Pickup and Delivery Problem* (PDP). So the *depot* searches, based on these decisions, other solutions for the *PDP*. If a solution is found that is better than the initial one, it replaces the initial solution. This is done, by telling each *vehicle*, which of its newly generated plans it has to take as its new one. To transmit this information, the *number* of the *round*, whose resulting plan should be taken by the *vehicle*, is sent. For example if *Round 3* is sent, then the *vehicle* knows, that the plan it created at the *third* round, is it's result of this protocol run. After that, the protocol ends.

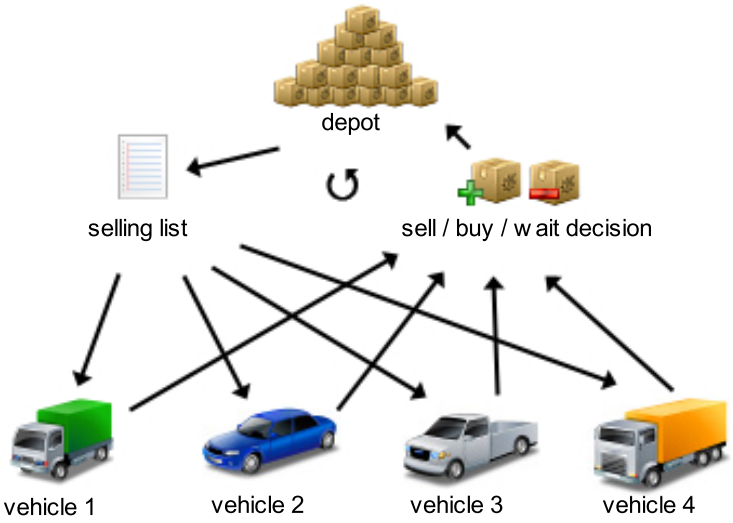


Figure 3.19: *Step 2: Sell-And-Buy Phase is done iteratively, until an certain threshold.*

3.4.3.3 Using the Trading Graph

This subsection describes, how the buying and selling actions, which the *vehicle* make during the *STP* rounds, are used by the *depot*. Furthermore, it describes, how the *depot* finds a solution for the *PDP*, according to these actions. Each round is called a *decision level*, since each *vehicle* has to make a decision in a round. This decision is sent to the *depot*, and contains the following information

- if a *buying*, or *selling* action should be performed at this level. Or *wait* (no operation) if the chosen action could not be performed.
- which task should be the object of the buying or selling action (nothing if a wait action must be performed).
- how much costs the buying would cause, or how much could be gained by selling the selected order (nothing, if a nop action must be performed).

These information can be used to create buying and selling *nodes*. Furthermore, the *nodes* are connected as follows. For each trading action, there is an *edge* between the buying and selling nodes that correspond to the buying and selling action of the trading action.

So an edge between two nodes indicates that the buying and selling object in the two actions is the same. Thus an edge tells, that in the buying nodes' action, the object is *bought* and in the selling nodes' action *sold*. The weight

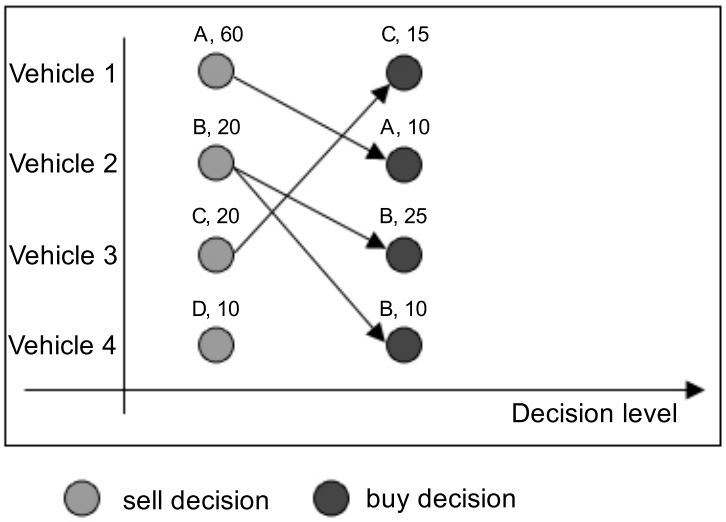


Figure 3.20: *Step 3: Calculate Trading Graph according to the decisions from the vehicles and find node edges.*

of an edge is the difference between the *gains* and the costs of the nodes, that are connected by this edge. So let $gain_s$ be the gain of a selling node s and $costs_b$ the costs of a buying node b. Then the weight w of the edge, which connects s and b, is given by

Definition 5 $w = gain_s$ - $costs_b$

The set of all these nodes and edges builds up a graph, which contains all trading actions between the *vehicles* and therefore is called a *Trading Graph* (TG). A *TG* is organised as a $m * n$ matrix, where m is the number of *vehicles* that participate in the STP, and n is the number of decision levels that are performed during the STP.

Figure 3.20 shows the *TG* that is generated during two decision levels of our example. In this *TG*, one can see that *Vehicle 1* sells order *A* and saves 60 cost units at the first decision level, while *Vehicle 2* buys order *A* at the second decision level, which causes 10 cost units. Thus, if *Vehicle 1* sells *A* to *Vehicle 2* this action would causes a deficit of 50 cost units to the resulting solution. Another important thing that can be noticed is the fact, that *Vehicle 2* can buy *A* with costs of 10, because it sold order *C* on the first decision level (see Definition 6., item (ii)). This shows that each decision of a *vehicle* on the n-th decision level depends on the decisions it has made on the previous levels. Thus an algorithm, which searches a legal

set of selling and buying actions, that ends up in a better solution for the underlying *PDP*, has to consider this aspect.

A Subgraph of a *TG*, consisting of so called *matched* nodes (nodes whose actions are selected to be realised for a new solution), is called a *Trading Match* (TM), if and only if:

Definition 6

(i) for each matched node there is exactly one other node in the TM which is connected to it by an edge. This means that for each buying node there is a corresponding and connected selling node in the TM and for each selling node there is exactly one corresponding and connected buying node in the TM.

(ii) for each matched node at decision level x, all other nodes of this vehicle, which are positioned on an earlier decision level y (where $y < x$), are also matched and so are part of the TM.

Waiting nodes are not affected by these conditions. That is the case, since *waiting* nodes represent no performable actions and thus do not change the plans of the *vehicles*. So it is not necessary, to match these nodes. But if the conditions are fulfilled for buying and selling nodes, the *TM* contains the actions that have to be performed by the *vehicles*, in order to create another solution for the *PDP*. The quality of such a solution can be derived from the *TM*, by summing up the weights of the edges, which connect the matched nodes.

So let $\{e_0, \ldots, e_k\}$ be the set of edges, that connect the nodes of the *TM* and w_i the weight of the i-th edge out of this set. Then the weight W_{TM} of this match is given by

Definition 7

$$W_{TM} = \sum_{i=0}^{k} w_i$$

If this sum is positive, then a better global solution for the *PDP* is found, since the saved costs by the selling actions are greater than the additional costs of the buying actions.

Figure 3.21 shows the *TM*s that could be found in the *TG* of Figure 3.20. Here we have included trading matches with the weight for each edge, and

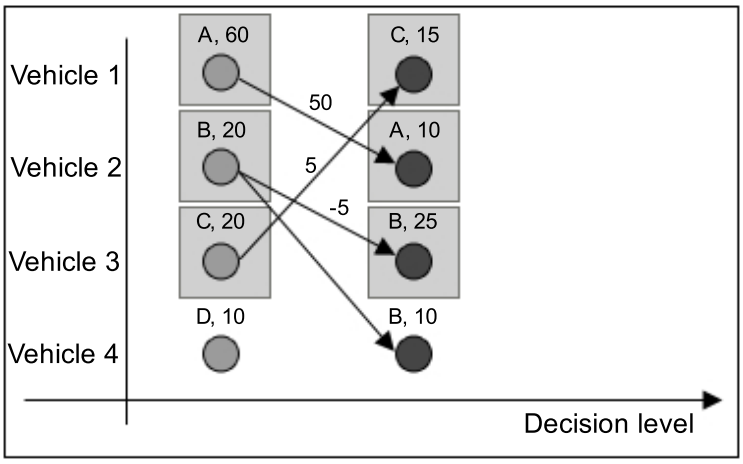


Figure 3.21: *Step 4: Search Trading Match Phase.*

when we calculate total weight of the matches, we get sum of 50. Since this sum is positive, we have a better global solution for the *PDP* compared with the initial tour plan.

You may ask yourself, shouldn't we get better improvement if *Vehicle 2* sells order *B* with cost of 20 minutes to Vehicle 4? And you are probably right, we should actually get positive weight of 10 instead of -5. But since no other vehicles have interest in buying order D from *Vehicle 4*, which he is selling in his first (and previous) decision round, according to Definition 6 we got no Trading Match.

In conclusion, if *Vehicle 1* would take order *C* with cost of 15 minutes, this would give the global improvement for the whole tours of 50 minutes from the original tour plan.

To generate the plans, that correspond to the better solution, the *Depot* has to tell *Vehicle 1*, *Vehicle 2* and *Vehicle 3* to do all the actions to the second decision level and the protocol ends.

3.4.3.4 Dynamic Scheduling Problems

All these steps work fine, as long as a complete *PDP* given to the system and this problem has to be solved and optimised. But problems occur, if additional orders are given to the system, while the optimization process is already running. Since the result of the optimization does not include this new order, it cannot be scheduled while an optimization process is running. This would result in data inconsistencies. To handle this problem, there are two options that were considered.

The first way to get rid of this problem, is not to schedule the new order until the optimization is done. This would decrease the efficiency of the

system. In addition to this, the time the optimization process is running, could be several seconds.

The second way to solve this challenge would be to stop the optimization process and to use the result that was up to now the best if it proves to be an improvement. Then the new order can be scheduled and the optimization process can start again and can also include the new order.

3.4.4 Summary

So far, we have introduced two protocols, the *Contract Net Protocol* and *Simulated Trading Protocol*, where both relies on communication between several entities. For our industrial use case, we will use the *Contract Net Protocol* to initiate the communication, while the *Simulated Trading Protocol* for the optimization. Figure 3.22 summaries the interaction flow between the *Planner* and *Heat* agents in the heat and sequence optimization inside supply chain of steel production.

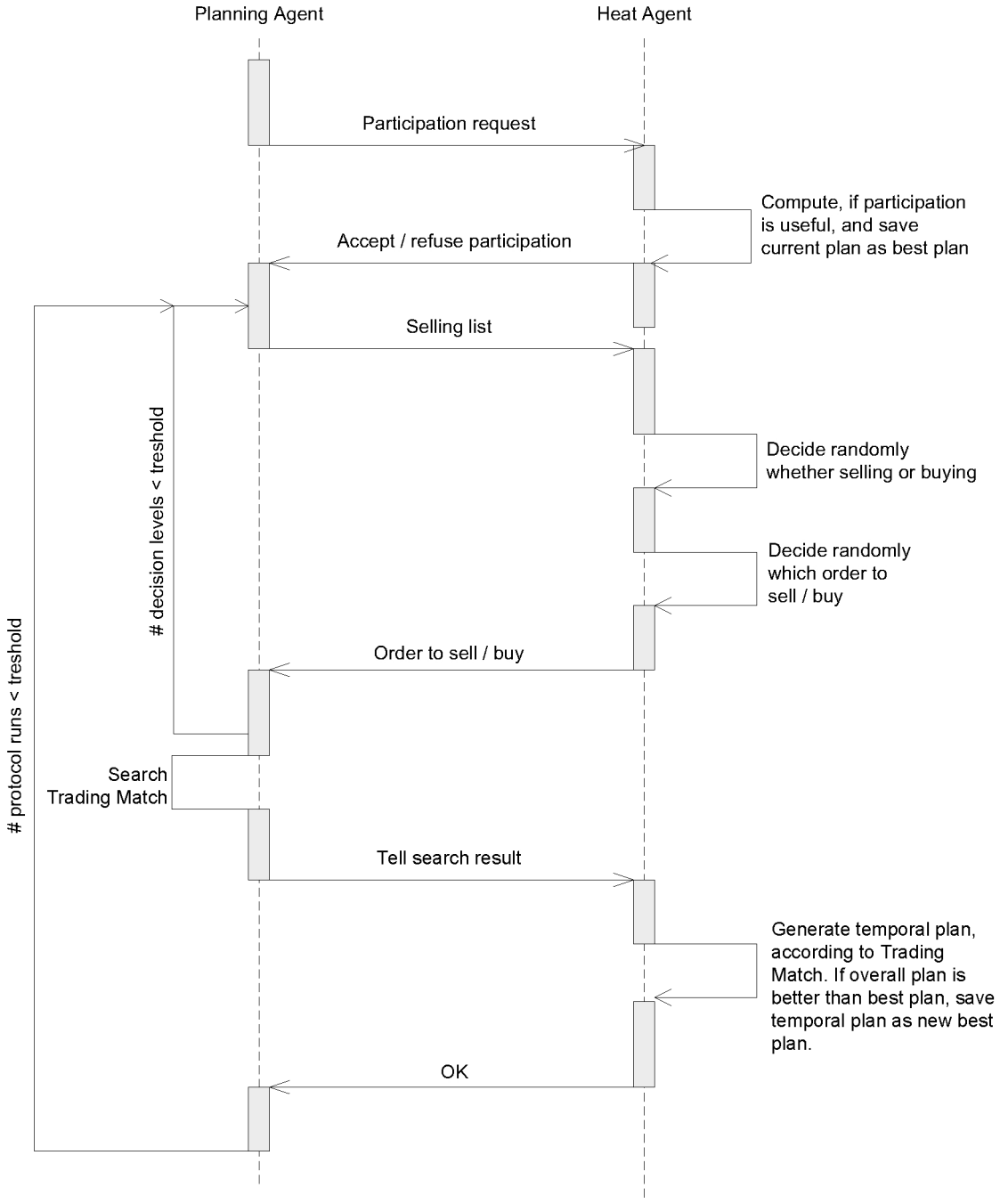


Figure 3.22: *Use of Simulated Trading Protocol together with Contract Net Protocol in Heat and Sequence Optimization inside the Supply Chain of Steel Production.*

Chapter 4

UPMS-a: Extensions for Interaction Protocols

> Any fool can make things bigger, more complex, and more violent. It takes a touch of genius - and a lot of courage - to move in the opposite direction.
>
> Albert Einstein

So far we have been introduced through the chapters about motivation and background, problem analyses and technologies related to our work. Here we are introducing the contribution and main work of this thesis.

Our work is influenced and based on Øystein Haugen's paper [70] on the very same theme. We will briefly present this work, and later apply it to the use case scenario in Chapter 5, as proof of concept.

We start with outlining the challenges in existing version of UML by beginning with model the *FIPA Contract Net Protocol*(CNP) in Section 4.1. Because the ability to multicast messages is one feature that is lacking in UML, we will show how some very small enhancements of UML make the language more suited to express agent protocols. First we start with introducing configurations with subsets in 4.2 and subset notation of messages in 4.3. Thereafter we will explain the semantics of the multicasting and the iterator-clause in Section 4.4. We also realized that current version of UML is not providing any guidelines for how timers could be modeled. Timers are important within interaction protocols and that to prevent deadlocks. Therefore, we will present best-of-practice alternatives to realize such mechanism, in Section 4.5.

4.1 Contract Net Protocol modeled with current UML

As we explained in Section 3.4.2, the *CNP* shows how an initiator sends out a number of *call for proposals* (cfp) to a set of participants. Some of these participants will *refuse* the call, while others may come up with a *proposal*. The initiator will then consider the proposals and *reject* some, and *accept* some. For those that are accepted, there are three different final results sent back to the initiator.

The FIPA protocol depicted in Figure 3.17 is not a valid UML 2 diagram. It is described in a dialect of sequence diagrams using extensions to *Agent UML*, suggested by Huget [45]. The notation used has included multiplicity on the messages on the ends to show how many message instances the diagram really describes. This approach functions well on an informal level, but is not sufficient if we want to describe what happens with every participant. There is nothing in the FIPA diagram that says anything about which participants are involved in which sequence of messages. E.g. there is nothing in the diagram that expresses that the *reject-proposal* messages are sent only to participants that earlier sent proposals back to the initiator. This is obvious to the human reader, but is nowhere defined.

So let us take a look at how we could realize CNP in existing UML.

4.1.1 The Single Participant Approach

One approach is to define a UML *collaboration* (see Figure 4.1) where there is one initiator and one participant. That participant has all the abilities of any participant, but of course for one single case, the participant is only one person.

We believe that most UML designers are not familiar with use of UML Collaborations, so we choose to include definition noted in [61, page 168-171]:

> **Definition 8** *A collaboration describes a structure of collaborating elements (roles), each performing a specialized function, which collectively accomplish some desired functionality. Its primary purpose is to explain how a system works and, therefore, it typically only incorporates those aspects of reality that are deemed relevant to the explanation. Thus, details, such as the identity or precise class of the actual participating instances are suppressed.*

A collaboration is shown as a dashed ellipse icon containing the name of the collaboration. The internal structure of a collaboration as comprised

by roles and connectors, may be shown in a compartment within the dashed ellipse icon. Alternatively, a composite structure diagram can be used.

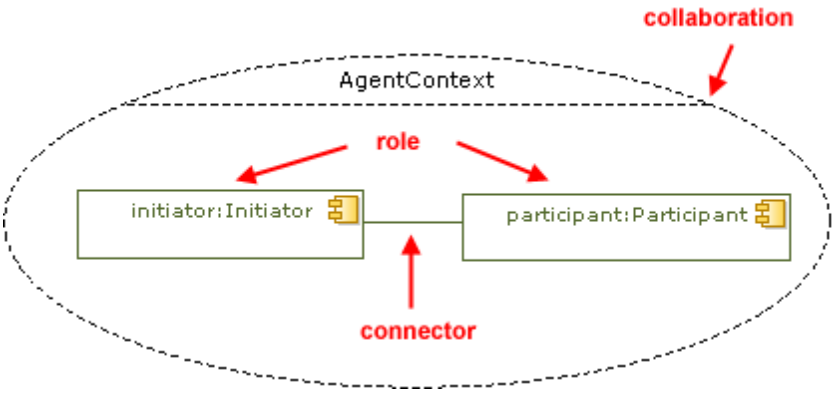


Figure 4.1: *Agent Context for the single general participant.*

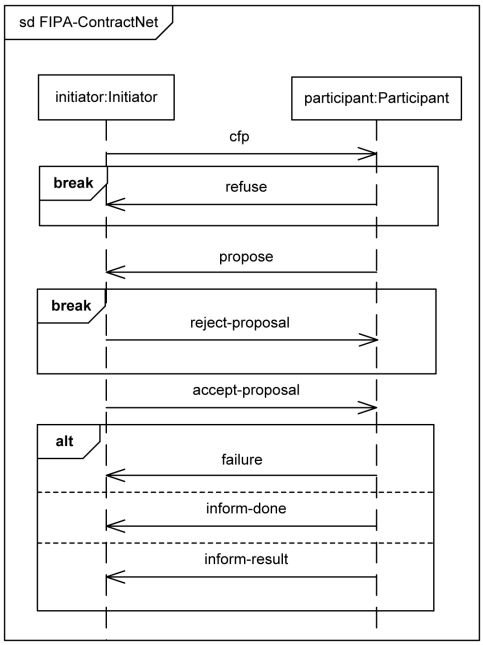


Figure 4.2: *UML FIPA protocol for the general participant.*

We apply the combined fragments of UML 2 sequence diagrams to define the different options that are open to the participant.

In Figure 4.2 we apply the *break-fragment* [61, page 467-472] which means that either the fragment contents happen or the rest of the diagram (actually the rest of the enclosing interaction fragment) happens. Thus we are able to express that for a participant that *refuses* the call for proposal, there will be

no continuation of the protocol. We notice also that the use of the **break**-fragments and the final **alt**-fragment (showing three different alternatives) makes the diagram more compact than the original, and in fact is a lot more precise. However, we still have the problem that the approach misses the point that the initiator sends calls to many potential participants and that he in fact must cope with many replies. The diagram shown in Figure 4.2 shows a very limited view, the situation as seen from the general participant.

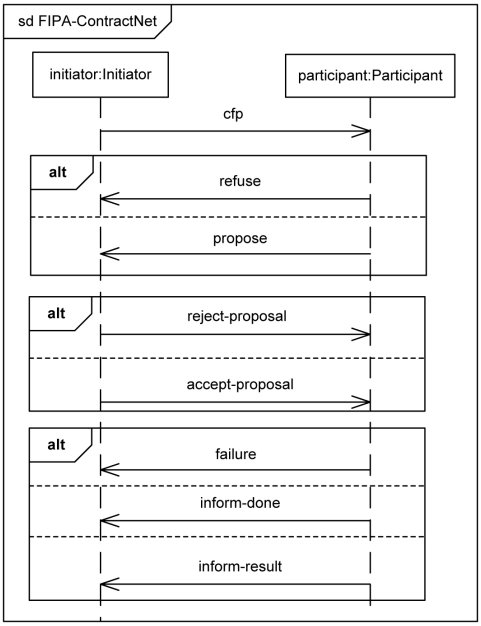


Figure 4.3: *UML FIPA protocol for the general participant, expressed with only alt-fragments.*

Break-fragment is one of the new concepts in UML 2 and many designers have not used it before. To illustrate how the same sequence could be modeled with more well-known concept, we can replace each break-fragment with alt-fragment as depicted in Figure 4.3. However, the first approach gives more compact and precise diagrams, so we will continue with use of break-fragments in following sequence diagrams.

4.1.2 The Multiple Participant Approach

Our next attempt is intended to express that there are different participants with different situations relative to the initiative described in the protocol. Rather than defining one participant that is fully general, we now describe one participant for each distinct situation. In our collaboration Figure 4.4 we define a set of participants with multiplicity.

We apply a notation in the lifeline header to select one object from a set by a selector. The selector in this case is only a symbolic name serving as an

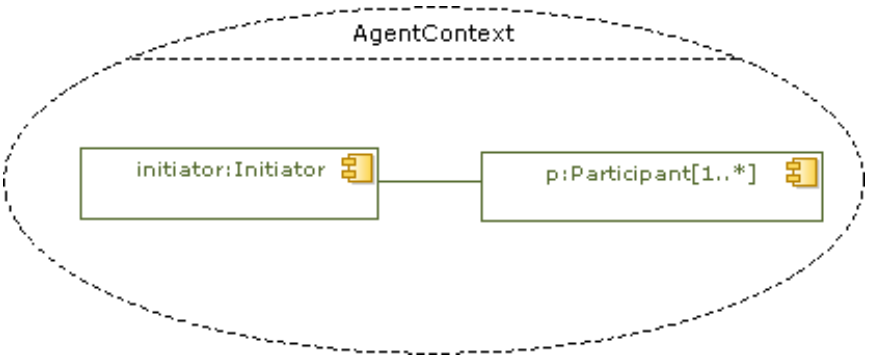


Figure 4.4: *Agent Context with set of participants.*

index indicating what situation that given participant object is representing.

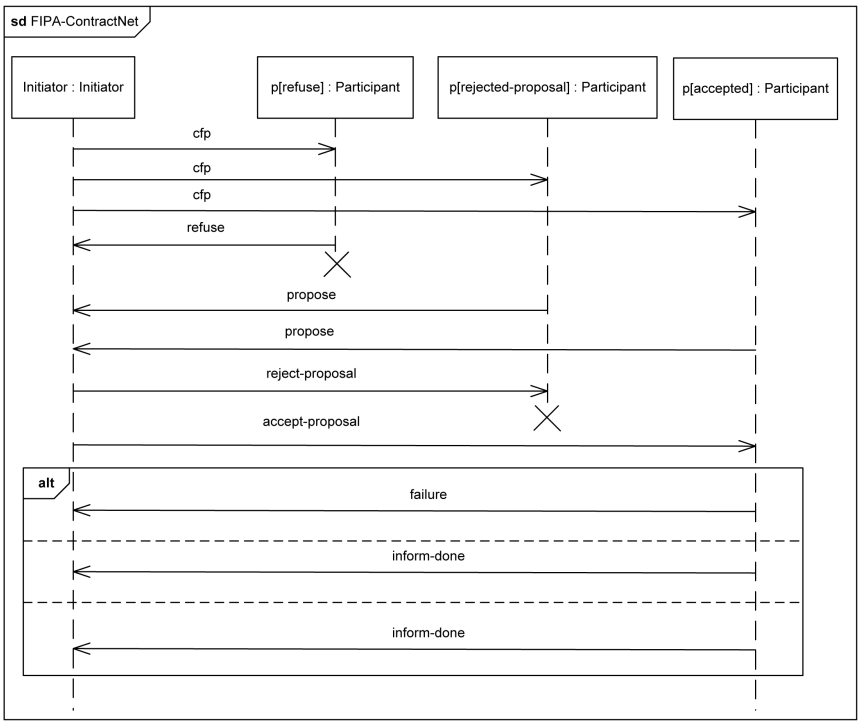


Figure 4.5: *UML FIPA protocol for typical participants.*

In Figure 4.5 we have one lifeline for each of the typical situations a participant may be in. In this way we visualize more directly that there are several participants that the initiator must relate to. We are not able, however, to describe the multiplicities of each of the subgroups. This approach fails to give the impression that the initiator has the same approach to all

participants in the first place. It also shows a situation where the initiator handles the different typical participants in *strict sequence*. This is by no means illegal as this diagram is not necessarily intended to define all possible traces of the protocol. Sequence diagrams show *possible* runs, but seldom all possible runs. Still we may want to express that there is nothing that prevents the initiator from handling the return from an accepting participant before the return from a refusing participant. We can achieve that by introducing such standard constructs as coregions which express that events may come in any order.

4.2 Introducing Configurations With Subsets

The problem with the approach of typical participants was that the sets from which these typical lifelines were selected, were not properly described. Their multiplicity and internal relationships were not defined. We remedy this by introducing configurations with subsets. In fact this is available in UML 2 already, but seldom used in modeling with composite structures, while the UML 2 metamodel has very many subset declarations on association ends in class diagrams. Subsets are constraints on a class property indicating that the defined property is a subset of some other property defined in a superclass of the encloser. An example (Figure 4.6) will make this clearer.

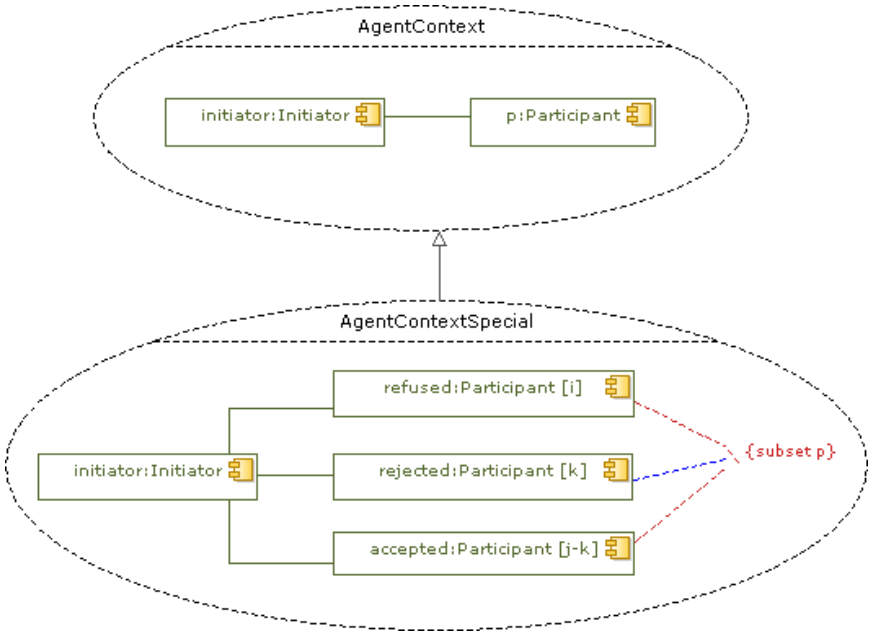


Figure 4.6: *Agent Context with specialization.*

In *AgentContext* we define one total set of participants p with multiplic-

ity *m*. *AgentContextSpecial* is a specialization of the general *AgentContext* where we have defined three subsets of p and these subsets are given new names *refused, rejected* and *accepted*. What this does is to keep the information that all objects of these sets are still contained in the original p, but that they may have added capabilities or situations. In fact it would be natural in our context to have *three* layers of specialization. The first distinction goes between those that refuse and those that propose. The second distinction defines subsets within those that propose, namely between the rejected and the accepted. We flattened the two lower subclasses for illustration simplicity.

The behavior definition given in Figure 4.7 is structurally equivalent to that of Figure 4.5, but the names of the lifelines reflect the subsets. Within each of the subsets we apply the single lifeline approach as every object within the subset should exhibit the same behavior.

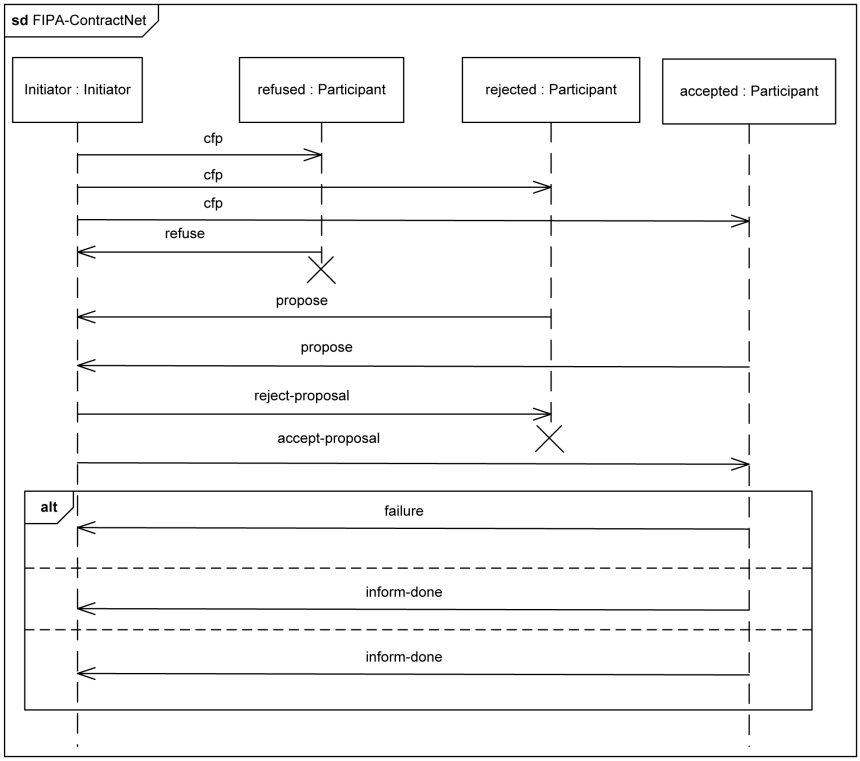


Figure 4.7: *UML FIPA protocol for typical participants, with subsets.*

Notice that we have not applied any constructs that are not in UML 2 already, but we have in this latest approach applied a description technique that is not very common. The technique has been presented with additions by Haugen and Møller-Pedersen in [73].

4.3 Introducing Subset Notation on Messages

We have found that the subset construct, that can be effectively applied to composite structures as shown in Figure 4.6, are useful also as identifiers in sequence diagrams as shown in Figure 4.7. But the latter still fails to capture properly the distinction between showing one typical instance of the set (as depicted first in Figure 4.5) and expressing that the interaction actually holds for every instance of the set.

Applying multiplicities to the messages as suggested by the FIPA protocol standard (Figure 3.17) does not quite express this precisely as this is not a matter of *numbers* only, but rather a matter of *subsets*. That is why we suggest to introduce a subset notation for messages that very much corresponds to the FIPA protocol standard.

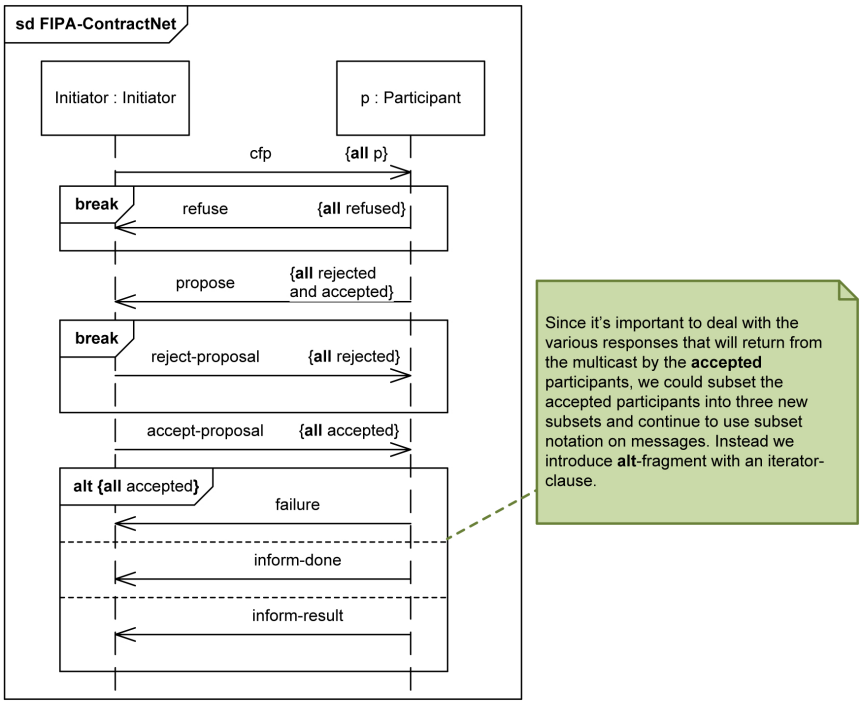


Figure 4.8: *UML FIPA protocol with subset message notation.*

The notation is simple. Attached to one (or both) ends of a message there is a constraint that has the keyword **all** followed by a part name. That part must be a subset of the part represented by the lifeline on the message end with the all constraint. In Figure 4.8 we have that *refused, rejected* and *accepted* are all subsets of p. Notice also that we have reached the same kind of compactness in our description that we had with our first approach with the general participant in Figure 4.2.

Informally the meaning of this notation is the same as described by the

FIPA protocol. Take e.g. the *refuse* message from *p* lifeline to *initiator* where the sending has attached the constraint {**all** refused}. This obviously is intended to mean that there is one message from every member of the refused set to the initiator.

Likewise the *reject-proposal* message with {**all** rejected} constraint on its receiving side intends to describe one message sent from *initiator* to every member of the rejected subset of *p*.

But, we need one construct more. It is not sufficient only to be able to define multicasting of messages to or from a given subset. The problem is not to define how to send a bunch of messages out, the problem lies in defining how to deal with the various responses that will return from that multicast. In Figure 4.8 we see how the *propose* message or the *refuse* message are responses to the initial *cfp*. We find that defining the *refused*, and *rejected* ∪ *accepted* subsets is a fruitful way to define the different alternatives. We could continue to subset the participants into smaller and smaller subsets, but in the end we choose a different variant. We define a combined alt-fragment with an iterator-clause. The "**alt** {**all** accepted}" fragment iterates over all participants of the accepted subset meaning that every accepted participant has that choice between sending back *failure, inform-done* or *inform-result.*

4.4 The Semantics of the Multicasting and the Iterator-Clause

We define the semantics of our proposed constructs as *shorthands.* We define a transformation procedure that transforms a diagram with multicasts and iterators into a standard UML 2 sequence diagram. The transformation should not really be carried out because it assumes creating lifelines for all objects of all parts in the composite structure.

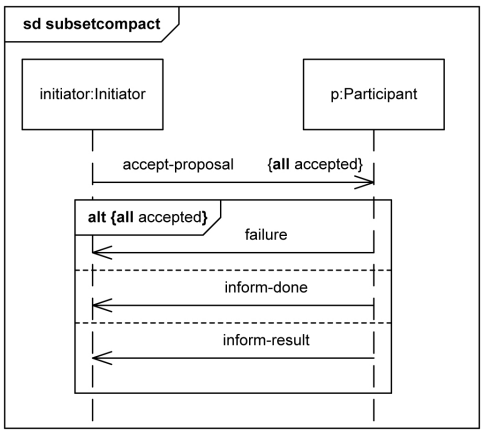


Figure 4.9: *UML FIPA protocol, compact version of multicasting and iteration.*

The sequence diagram in Figure 4.9 shows a compact version of the protocol isolating only the multicast and the iterator constructs.

Assume that the *accepted* subset of p only contains two participants, the expansion of the constructs result in the sequence diagram in Figure 4.10. The procedure for transforming multicasting has three steps:

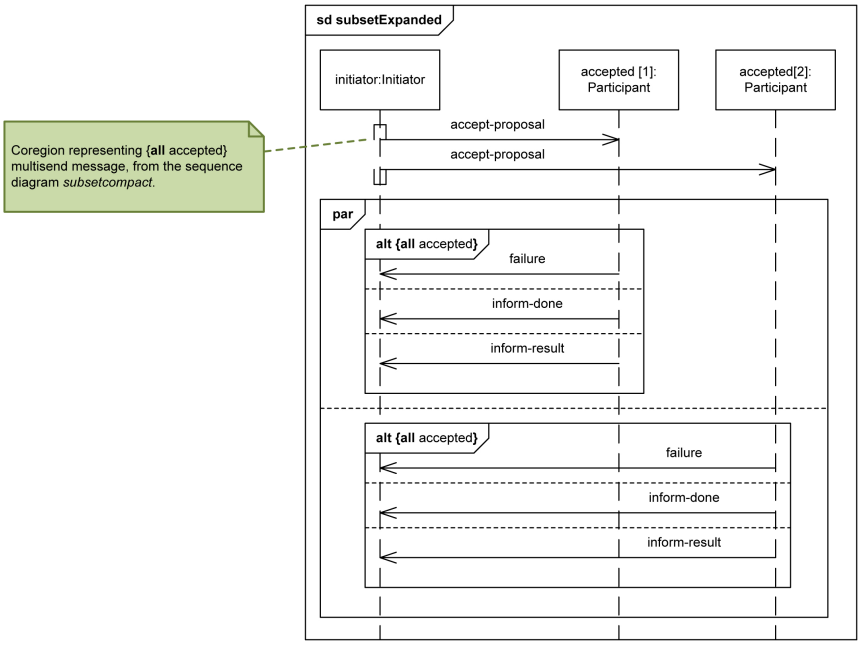


Figure 4.10: *UML FIPA protocol, expanded version of multicasting and iteration.*

1. Create one lifeline for each object in the subset mentioned in the **all**-constraint.
2. Replicate the message for every created lifeline.
3. Contain the multiple messages sends (or correspondingly receives) in a coregion indicating that the sending / receiving may come in any order.

In the situation where the **all**-constraint is used on both ends of a message, it is necessary to replicate the message for every pair of lifelines from the two lifeline sets involved (on either side of the message). On every lifeline there will be a coregion.

The iterator has a similar definition and in fact it can be seen as the more general of the two procedures since the multicasting can be described as if the message was contained in a special combined fragment with iterator.

1. Create one lifeline for each object in the subset mentioned in the iterator clause.

2. Replicate the combined fragment for every created lifeline.

3. Contain each created combined fragment in an operand of an enclosing par-combined fragment.

The *par-fragment* is defined in UML 2 as a parallel merge operator [61, page 468] that will merge in all possible ways the sequences of the operands.

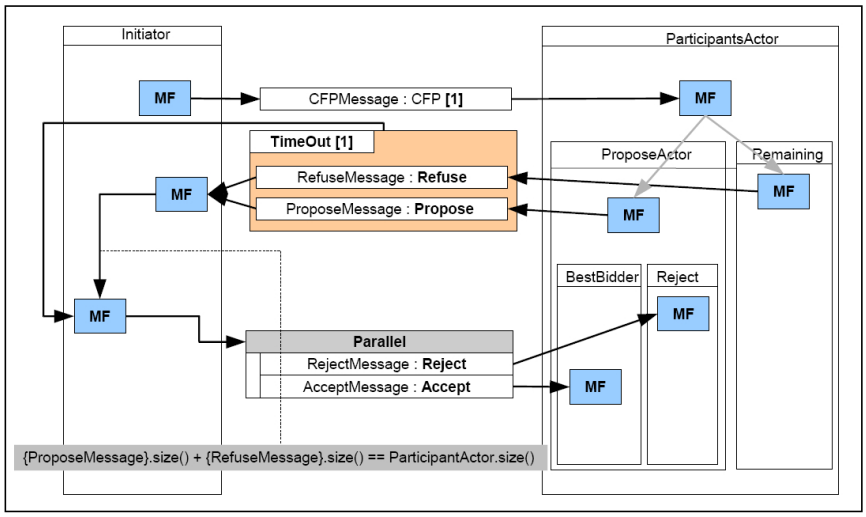


Figure 4.11: *Multicasting in FIPA Contract Net Protocol, expressed with Pim4Agents (see Section 3.3.3).*

That we can define our new constructs as shorthands that may be transformed into standard sequence diagrams means that we have not introduced anything that obstructs the good semantic properties of sequence diagrams such as compositionality. By compositionality we mean that the sequence diagrams can be refined piecewise and we can be certain that the result when putting these pieces back together is a refinement of the original. Using the STAIRS approach [72; 71; 40] we formalize this by a trace semantics where we can show that sequence diagrams are monotonic with respect to refinement for most of the standard operators such as *alt, seq, loop, par* [36].

This definition of multicast as explained, is not the first attempt to define multicasting in the area of sequence diagrams. Helouet made a definition in [41] which was combined with the ITU version Z.120 where he defines multicast groups that in some way resembles our subsets. *DFKI* has also very similar approach for solving the multicasting in their interaction diagrams of the *Pim4Agent* metamodel (see Section 3.3.3).

As the Figure 4.11 illustrates, we see that definition of multicast groups as well is used. There are two main actors, *Initiator* and *ParticipantsActor*, where the *Initiator* sends *CFP* message to *ParticipantsActor* and the *messageflow* (MF) continues further to the subsets *ProposeActor* and *Remaining*

(or called *refused* in Figure 4.8). Then these two groups replies with either *RefuseMessage* or *ProposeMessage*, which both are FIPA compliant, and at the end we can also see that *Parallel* concept is used to illustrate that *RejectMessage* and *AcceptMessage* goes in parallel to *BestBidder* and *Reject* participants.

Certainly, this definition divides participants into additional and smaller sub groups, but as mentioned earlier, this is possible with our approach as well. Furthermore we can see that this approach neither apply multiplicity or subset notation to the messages, but the containment of participant subsets inside participant have the semantics that this means "all" participants of certain type of subset.

4.5 Use of Timer in UML 2

In scenarios where two or more actors are waiting for each other to reply or finish their actions, it is important to prevent the possible deadlock that could occur if they never respond. Usually timer is used for this purpose, where the initiating actor is starting the timer when sending some request and if the response is not received within certain duration, timeout occurs and initiator can continue further on doing other tasks.

Our understanding of the *deadline* event used in FIPA CNP, Figure 3.17, is that this notation might be obvious to the human reader, but is nowhere defined. According to the diagram, it is hard to understand who is starting the timer and when. Neither do we see what triggers the *deadline* event that occurs after the *cfp* message. Haugen's approach for bringing multicast of messages in UML 2 sequence diagrams, as described in previous section, neither clarifies this further.

However, there are three alternative ways to model timers with UML 2, but rarely used and known by UML designers; i) custom classifier that represents timer, ii) use of SimpleTime model included in UML Superstructure and iii) create more sophisticated model of time provided by an appropriate UML profile. We will now give a short introduction to these three alternatives.

4.5.1 Custom Classifier Representing Timer

We can, with help of an classifier and some neat sequence diagram elements, build our own timers with UML 2. Sequence diagram in Figure 4.12 describes interaction between *Initiator* and particular *Participant*. The *Timer* lifeline is an classifier and could for instance be implemented as UML Class or Component. The timer is set by sending a message *startTimer* with the delay of some duration, in this case 5 time units, which is assigned to a parameter *d*. A variable *expire* is assigned the point in time the timer should

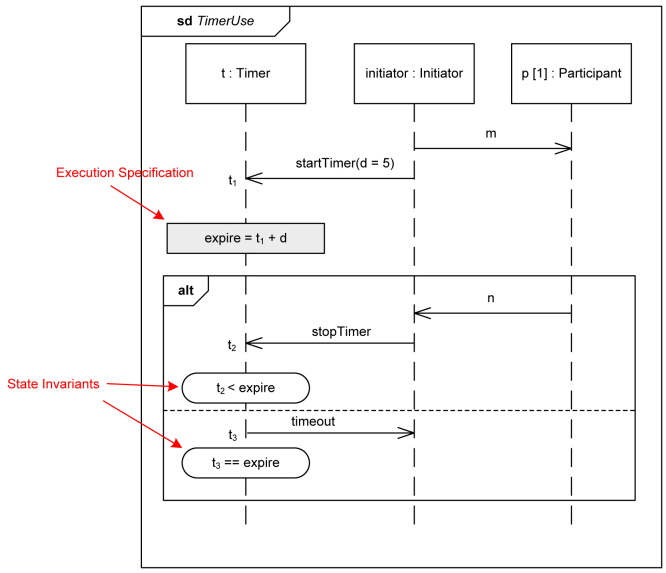


Figure 4.12: *Use of a Timer in UML 2, from [66, page 201].*

expire as the sum of the timestamp *t1* of the reception of the *startTimer*, and the delay *d*.

In the first alternative, *Initiator* sends a message *stopTimer* to *Timer*, and the reception of *stopTimer* on *t* is followed by a time constraint stating that the timestamp of the event should be less that the value of *expire*.

In the second alternative, the *Timer* sends a message timeout to *Initiator*, followed by a constraint specifying that the timestamp of transmission of timeout should be equal to the value stored in *expire*.

Specifying a timer in this way, as an independent entity, we place all the mechanism and implementation inside the Timer classifier, and can build more sophisticated model of timers. Disadvantages is that sequence diagrams become complex and harder to read.

4.5.2 SimpleTime Model in UML Superstructure

Most UML designers are not aware of that UML 2 is originally supporting constructs to help describing observation of time. Reason for this could probably be that almost none UML modeling tool is support this features and therefore it's rarely used.

The *SimpleTime* subpackage [61, page 423-454], of the *Common Behavior* package, adds metaclasses to represent time and durations as well as actions to observe the passing of time.

As the name tells, the simple model of time described in SimpleTime package, is intended as an approximation for situations where the more complex aspects of time and time measurement can safely be ignored. However,

SimpleTime is not explicitly introducing "timer" to UML, but instead provide support for representing *time* and *duration*.

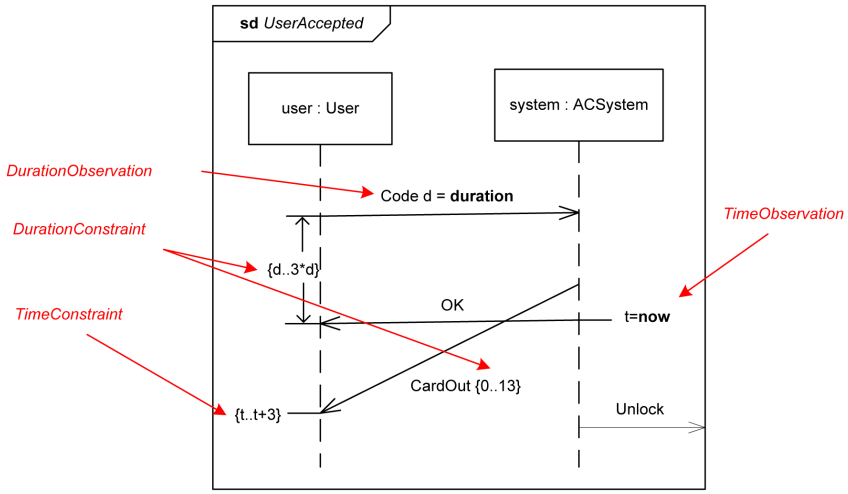
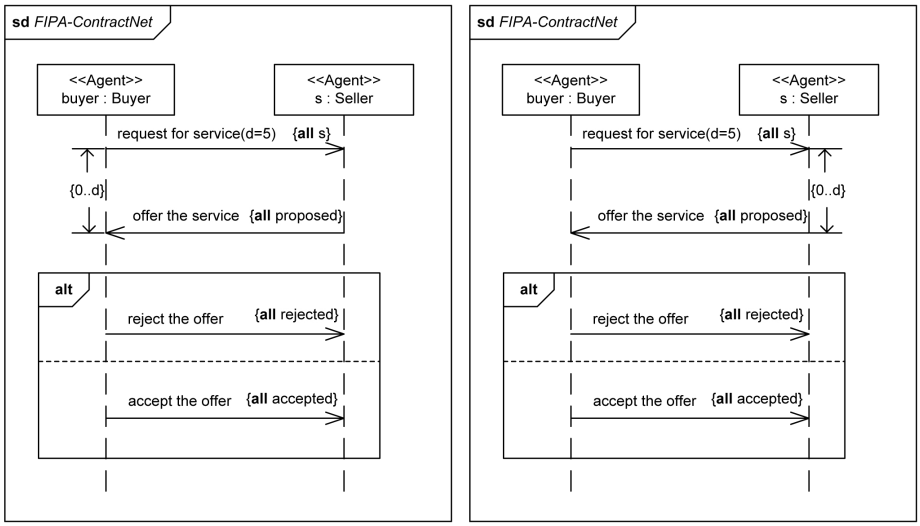


Figure 4.13: *Simpletime - sequence Diagram with time and timing concepts, from [61, page 513].*

The Sequence Diagram in Figure 4.13 shows how time and timing notation may be applied to describe time observation and timing constraints. The *User* sends a message *Code* and its *duration* is measured. The *ACSystem* will send two messages back to the *User*. *CardOut* is constrained to last between *0* and *13* time units. Furthermore the interval between the sending of *Code* and the reception of *OK* is constrained to last between *d* and *3*d*, where *d* is the measured duration of the *Code* signal. We also notice the observation of the time point *t* at the sending of *OK* and how this is used to constrain the time point of the reception of *CardOut*.

The *DurationConstraint* could be applied to CNP as demonstrated in Figure 4.14, which is simplified CNP. The figure shows two diagrams, almost equal to each other, describing scenario where one *buyer* wants to buy a service and sends *request for service*(cfp) to all potential *sellers*. After the offers are received, *buyer* picks out one or many *sellers* to provide this service and rejects the others. Difference between diagram *a* and *b* is that we are describing deadlines for different actors. In *a* we describe deadline for **receiving** the offers at buyers side, while in *b* we describe deadline for **sending** offer from seller side. The upper arrow in DurationConstraint illustrates that the clock starts ticking when *request for service* message is sent (or received in diagram b). The lower arrow illustrates that the DurationInterval *0..d* is checked on the reception of *offer the service* message (or sending in diagram b). All traces where the constraints are violated are negative traces i.e., if they occur in practice the system has failed.

In principle, the SimpleTime is applicable to all behaviour constructs of



(a) Buyer deadline (b) Seller deadline

Figure 4.14: *DurationConstraint applied to simplified FIPA CNP.*

UML, for example sequence diagrams, state machines and activity diagrams. It is also more compact approach compared with one described in previous section. But since there is no or very little use of this package, there are also very few examples to relate to in order to get more practical understanding.

4.5.3 Timer Described with UML Profile

Finally, the third and last alternative way of describing timers with UML is to define your own UML Profile, or use existing one. One example is the *UML Testing Profile* [57]. This is OMG specification, based on UML 2.0, which defines a language for designing, visualizing, specifying, analyzing, constructing, and documenting the artifacts of test systems. It is a test modeling language that can be used with all major object and component technologies and applied to testing systems in various application domains.

Now, the interesting part of this profile is the *Time Concept* [57, page 29-36]. The *Simple Time* concepts of UML do not cover the full needs for test specification, and therefore the Testing Profile provides a small set of useful time concepts.

In Figure 4.15 we have described the same sequence diagram as in Figure 4.12, but with time concepts from *UML Testing Profile*. As you can notice, the *Timer* is now represented as messages and not lifeline as in the first diagram. Also *startTimer, stopTimer* and *timeout* messages has been replaced with symbols.

Timer is a predefined interface. *Timer* properties may only be owned by

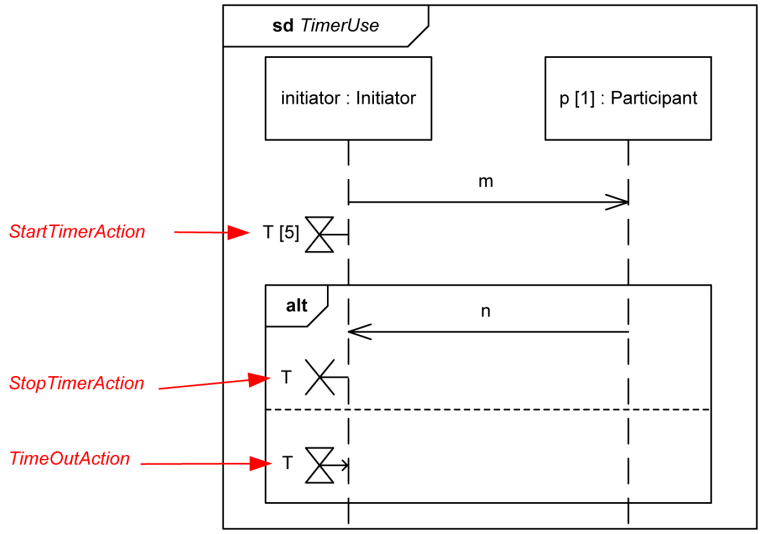


Figure 4.15: *UML Testing Profile: sequence diagram with time actions, from [66, page 201].*

active objects. An active class may own multiple timers. Operations like *start, stop* and *read* are defined for the timer interface.

By means of the *start()* operation, a timer may be started with a certain time value. The predefined time value of a timer is always positive. For example, "start Timer1(now+2.0)" means to start a timer and to stop it at latest in 2 time units, otherwise it expires.

With the *stop()* operation, an active timer can be stopped. The expiration time of an active timer can be retrieved by the *read()* operation. The timer attribute *isRunning* is a boolean value and indicates whether the timer is still active or not. When a timer expires after its predefined time, a special *timeout* message is generated automatically. It is sent immediately to the active class that owns the timer. A timeout is only allowed to be sent to the owning class of the timer.

When we tried to apply the UML Testing Profile to our case study, we found some legacy problems regarding UML version that was used in original profile. The specification claims that the profile is based on the UML version 2.0. But when we tried to define the profile from the scratch, with respect to the specification [57], we recognized that some metaclasses was missing (or replaced).

However, we believe that such concepts are fruitful and important with respect to the agent interaction protocols, and has proposed this as one mandatory requirement in the *Agent Metamodel and Profile* (AMP) RFP. We get more compact and easier to read diagrams, and maybe the biggest advantage, compared with the first approach, we can reuse it in several models without having to describe it over again.

4.6 Summary

In the previous sections of this chapter we described the architecture and design of UPMS-a. We started with outlining the challenges in UML describing agent interaction protocols and slightly followed with introducing configuration with subsets, in order to group participants and define their multiplicity and internal relationship.

Subset notation on messages were introduced in order to apply what participant subset(s) the messages is being sent to and from. New Iterator-clause were explained as support for iterating the multicasting

Finally we presented three possible approaches to define timer concepts with UML 2.

Here we conclude the architecture and design chapter and leave the description of the realization and implementation details to the next.

Chapter 5

UPMS-a: Realization and Implementation

> In theory, there is no difference
> between theory and practice.
> But in practice, there is.
>
> Yogi Berra

In this chapter, we want to provide realization and implementation of the UPMS-a proposal on the use case from Section 2.1.1. With this, we want to proof that presented solutions in previous chapter is really useful with the respect to the agent interaction protocols, and it is possible to be realized and solve the problems in use case scenario from Saarstahl.

5.1 Introduction

The implementation consists of a *Collaboration* describing the requirements for a service for purchasing orders with the focus on scheduling productions, and the *ServiceInterfaces* and *Participants* that fulfill this contract. Furthermore, the specification encompasses a class diagram that describes the relations between the involved entities and at least the models of STP which emphasize optimization in the steel production as described earlier in the Section 3.4.3.

The scenario exists of four different, but collaborating domains; Purchasing, Invoicing, Productions and Shipping. We will only cover two of these, the Purchasing and Productions. For both domains we will explain the steps from business view to implementation and service view, which is in according to the Service-Oriented Modeling Architectre (SOMA), described in Section 5.2.

In Section 5.3 we focus on the high level requirements of the purchasing the order scenario, where we will also spend some time on explaining the use of SOA-Pro stereotypes.

In Section 5.4 we will explain in details the scheduling productions, and also apply the UPMS-a we described in previous chapter. Finally, in Section 5.5 we summarize the realization and implementation of the case study.

5.2 SOMA: Service Oriented Modeling Architecture

As in all development projects, it is very helpful to have guidelines, patterns and methodology on how to proceed in the different steps of the development process. Unfortunately, the SOA-Pro submission doesn't recommend any for us, so we have to choose our own SOA methodology in order to model the purchase order scenario.

However, we believe that IBM's *Service Modeling Technique* known as SOMA [4] (Service Oriented Modeling Architecture), is providing us the needed guidelines for our purpose. This technique consists of three key steps **Identification, Specification** and **Realization** of Services, Components and Flows (see Figure 5.1).

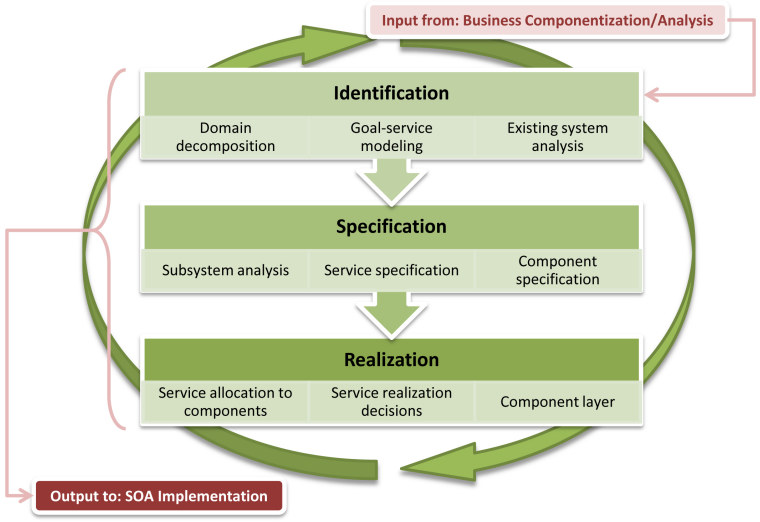


Figure 5.1: *Major activities in SOMA.*

In the Identification step, we will start with identifying the business domain, which is identifying the key business processes that supports purchasing the orders in the Saarstahl case study. This will be the input for identifying candidate services and data elements.

In the Specification step, we use identified artefacts from previous step and design them in details.

The last step, Realization, consists of realizing the functionality and collecting peaces together for the deployment.

5.3 Purchasing the Order

The requirements for processing purchase orders are captured in a Collaboration. Here we will not cover how this collaboration was determined from business requirements or business processes. The collaboration is then fulfilled by a number of collaborating Participants having needs and capabilities through *Requisitions* and *Services* specified by *ServiceInterfaces* identified by examining the collaboration. This collaboration is a formal, architecturally neutral specification of the requirements that could be fulfilled by some interacting service consumers and providers, without addressing any IT architecture or implementation concerns.

The *Process Purchase Order Process* collaboration in Figure 5.2 indicates there are four roles involved in processing purchase orders. The *orderProcessor* role coordinates the activities of the other roles in processing purchase orders. The types of these roles are the Interfaces shown in Figure 5.3. These *Interfaces* have *Operations* which represent the responsibilities or capabilities of these roles.

5.3.1 Service Identification

The next step in the development process is to examine the collaboration and identify services and participants necessary to fulfill the indicated requirements. Eventually a service provider will be designed and implemented that is capable of playing each role in the collaboration, and providing the services necessary to fulfill the responsibilities of that role.

Figure 5.4 shows a view of the service interfaces determined necessary to fulfill the requirements specified by the collaboration in Figure 5.2. Note that Interfaces are mapped to activity partitions (roles) in the activity diagram in Figure 5.2 and actions to Operations. This view simply identifies the service interfaces defining services that will fulfill the service contracts, the packages in which they are defined, and the anticipated dependencies between them.

5.3.2 Service Specification

The identified ServiceInterfaces must now be defined in details. A Service Interface defines an interface to a service: what consumers need to know to determine if a service's capabilities meet their needs and if so, how to use the service. A ServiceInterface also defines as what providers need to know in order to implement the service.

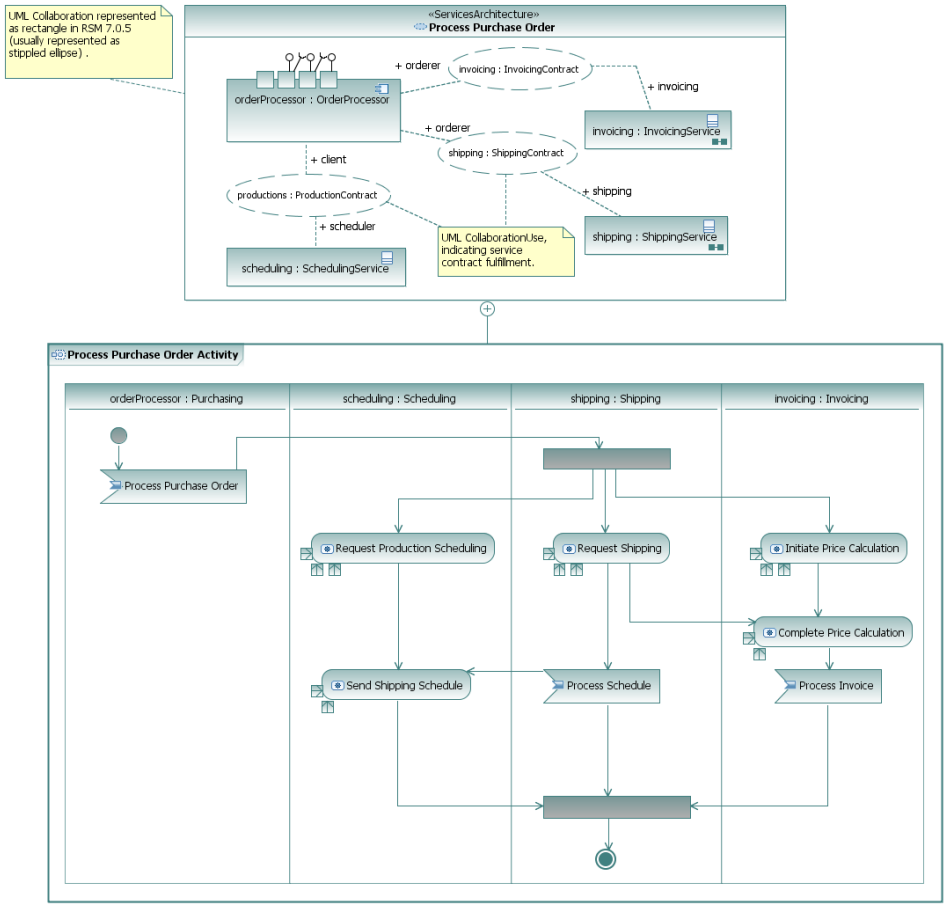


Figure 5.2: *High level overview and requirements of the purchase order scenario.*

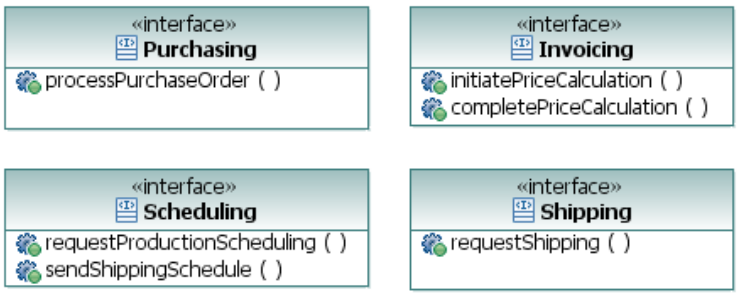


Figure 5.3: *Interfaces listing role responsibilities.*

Again, we choose only to focus on the *Purchasing* and the *Scheduling* (from Productions package) service interfaces. *Invoicing* and *Shipping* are out of the scope and will not be covered.

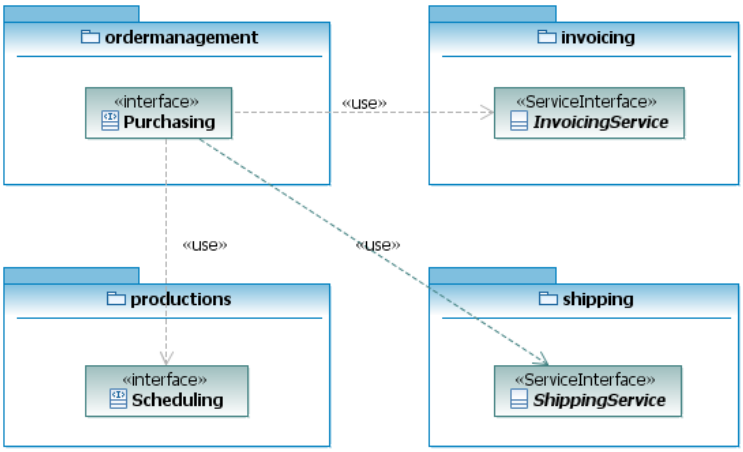


Figure 5.4: *Identified Service Interfaces.*

The requirement was to create a new purchasing service that uses the *Invoicing, Productions* and *Shipping services*, as determined in Figure 5.4 and according to the *Process Purchase Order* process. This will provide an implementation of the business process as choreography of a set of interacting service providers. Since this is such a simple service, no contract is required, and the service interface is a simple UML Interface providing a single capability as shown in Figure 5.5 (read more about Service Interfaces in Section 3.2.4.3).

The *processPurchaseOrder* operation has to in-parameters, containing customer related data and the purchased order. In return, the operation is providing invoice of the order.

Figure 5.5: *Purchasing service interface with capability as Operation.*

5.3.3 Service Realization

Part of architecting an SOA solution is to determine what participants will provide and consume what services, and how they do so. These consumers and providers must conform to any fulfilled contracts as well as the protocols defined by the service interfaces they provide or require. Each capability provided by a service participant must be implemented somehow.

Each capability (operation) will have a method (behavior) whose speci-

fication is the provided service operation. The design details of the service method can be specified using any Behavior: an Interaction, an Activity, StateMachine or OpaqueBehavior.

Often a service participant's internal structure consists of an assembly of parts representing other service providers, and the service methods will be implemented using their provided capabilities. We will not present participants and contracts for purchasing and production parts.

The purchase order processing services are specified by the Purchasing interface, and provided by the OrderProcessor provider as shown in Figure 5.6. This participant provides the Purchasing Service through its purchasing port.

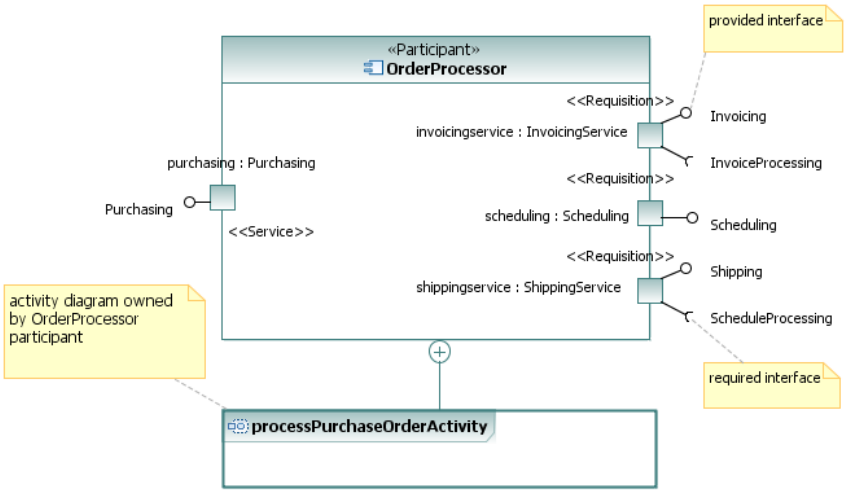


Figure 5.6: *The OrderProcessor Service Provider.*

The *OrderProcessor* participant also has Requisitions to three services: *invoicing, scheduling* and *shipping*. The providers of these services are used by the OrderProcessor component in order to implement its Services.

The *OrderProcessor* participant uses an Activity to model the design of the provided *processPurchaseOrder* service operation. The details for how this is done are shown in the internal structure of the OrderProcessor component providing the service as shown in Figure 5.7.

Each service operation provided by a service provider must be realized by either:

1. an ownedBehavior (Activity, Interaction, StateMachine, or OpaqueBehavior) that is the method of the service Operation, or

2. an AcceptEventAction (for asynchronous calls) or AcceptCallAction (for synchronous request/reply calls) in some Activity belonging to the component. This allows a single Activity to have more than one (generally) concurrent entry point controlling when the provider is able

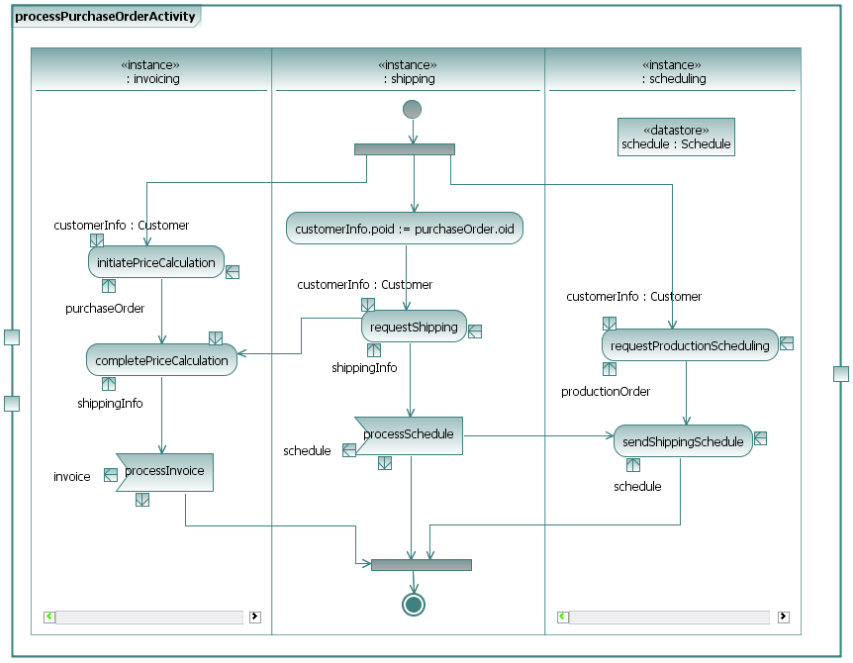


Figure 5.7: *The processPurchaseOrderActivity Service Operation Design*

to respond to an event or service invocation. These AcceptEventActions are usually used to handle callbacks for returning information from other asynchronous CallOperationActions.

Within the *OrderProcessor*, the *processPurchaseOrder* operation is the specification of the *processPurchaseOrderAvtivity* Activity which is an owned behavior of OrderProcessor.

This diagram corresponds very closely to the overview activity diagram with purchase order requirements that we specified in Figure 5.2. The *InvoiceProcessing* and *ShippingProcessing* service operations are realized through the *processInvoice* and *processSchedule* accept event actions in the process. The corresponding operations in the interfaces are denoted as «trigger» operations to indicate the ability to respond to *AcceptCallActions* (similar to receptions and *AcceptEventActions* where the trigger is a *SignalEvent*).

5.3.4 Assembling Services and Fulfilling Contracts

The OrderProcessor component is now complete. But there are two things left to do. First, new Participant must be created that connects service providers, capable of providing the OrderProcessor's required services, to the appropriate services. This will result in a deployable Participant that is capable of executing.

Second the OrderProcessor, and the other participants, needs to indicate that they fulfills the requirements specified in the collaboration shown in Figure 5.2.

The *OrderProcessor, Invoicer, Productions* and *Shipper* Participants are classifiers that define the services consumed and provided by those participants and how they are used and implemented. In order to use the providers, it is necessary to assembly instances of them in some context, and connect the consumer requisitions to the provider services through service channels.

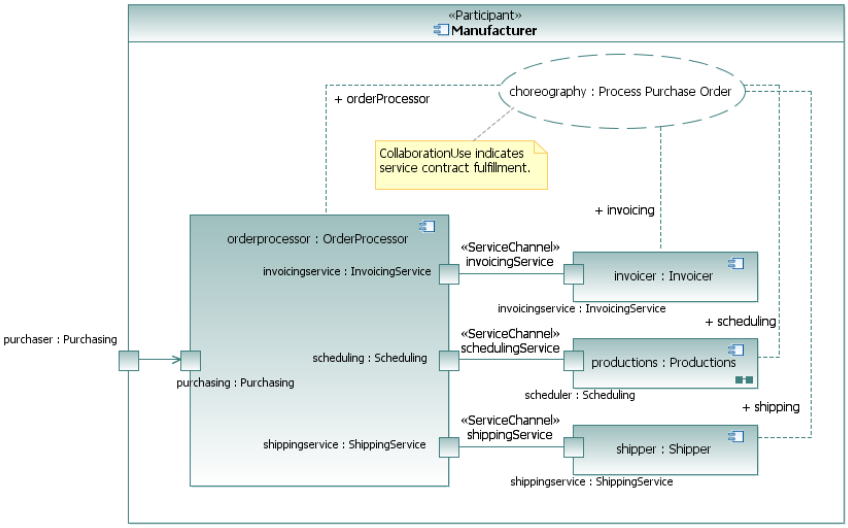


Figure 5.8: *Assembling the parts into a deployable subsystem, Manufacturer.*

The *Manufacturer* Participant shown in Figure 5.8 represents a complete component that connects the OrderProcessor service provider with other service providers that provide its required services. Figure 5.8 also shows how the Manufacturer participant provides the purchaser service by delegating to the purchasing service of the OrderProcessor.

Figure 5.2 describes the requirements for the OrderProcessor participant using a *Collaboration*. You may recognize that we are using the stereotype «ServiceArchitecture» and not «ServiceContract» of this collaboration. As we described in Section 3.2.4.6, Service Architecture is used to describes the roles of a set of Participants that provide and use services to achieve some mutual goal or implement a business process. Each service of a ServiceArchitecture is represented by the use of a ServiceContract bound to participant roles.

A *CollaborationUse* is added to the Manufacturer to indicate the service contract between the four participants. Service contract fulfillment are indicated by use of CollaborationUse.

The CollaborationUse, called requirements, is an instance of the Purchase

Order Process Collaboration (from Figure 5.2. This specifies that the participants fulfills the Purchase Order Process requirements. The role bindings indicate which role the parts of the service Participants plays in the collaboration. The *orderProcessor* part is bound to the orderProcessor role in the ServicesArchitecture. This part is capable of playing the role because it has the same type as the role in the architecture. The *invoicer* part is bound to the invoicing role of the ServicesArchitecture. This part is capable of playing this role because it provides a Service whose ServiceInterface is the same as the role type in the ServicesArchitecture. The *scheduling* and *shipping* roles are similar.

The *Manufacturer* Participant is now complete and ready to be deployed. It has specific instances of all required service providers necessary to fully implement the *processPurchaseOrder* service. Once deployed, other service consumers can bind to the order processor component and invoke the service operation.

In next sections we will go in details how the Production part is designed and implemented.

5.4 Production and Planning

In previous sections we identified overall requirements, and the collaboration between core participants through well defined interfaces. In this section we will explain the Productions part more in details, and apply UPMS-a proposals in creation and optimization of heats and sequences.

5.4.1 Service Identification

In Figures 5.2 we identified high level requirements for purchasing the order. This helped us to discovering the Productions role, which we have refined in more details in Figure 5.9. Two new roles are discovered for the planning part, *ProductionsPlanning* and *ProductionsUnitPlanning* (see Figure 5.10). Note that there are one or many (1..*) instances of the ProductionsUnitPlanning in the ProductionsPlanningContract. We have placed this multiplicity in the properties of the part, but choose to attach a Note in order to show this graphically in the diagram.

The ProductionsPlanning role is used to initiate and coordinate the execution of the STP, while the *ProductionsUnitPlanning* role decide whether to buy or sell OrderPositions in order to optimize the local plans (*computeNextAction* operation).

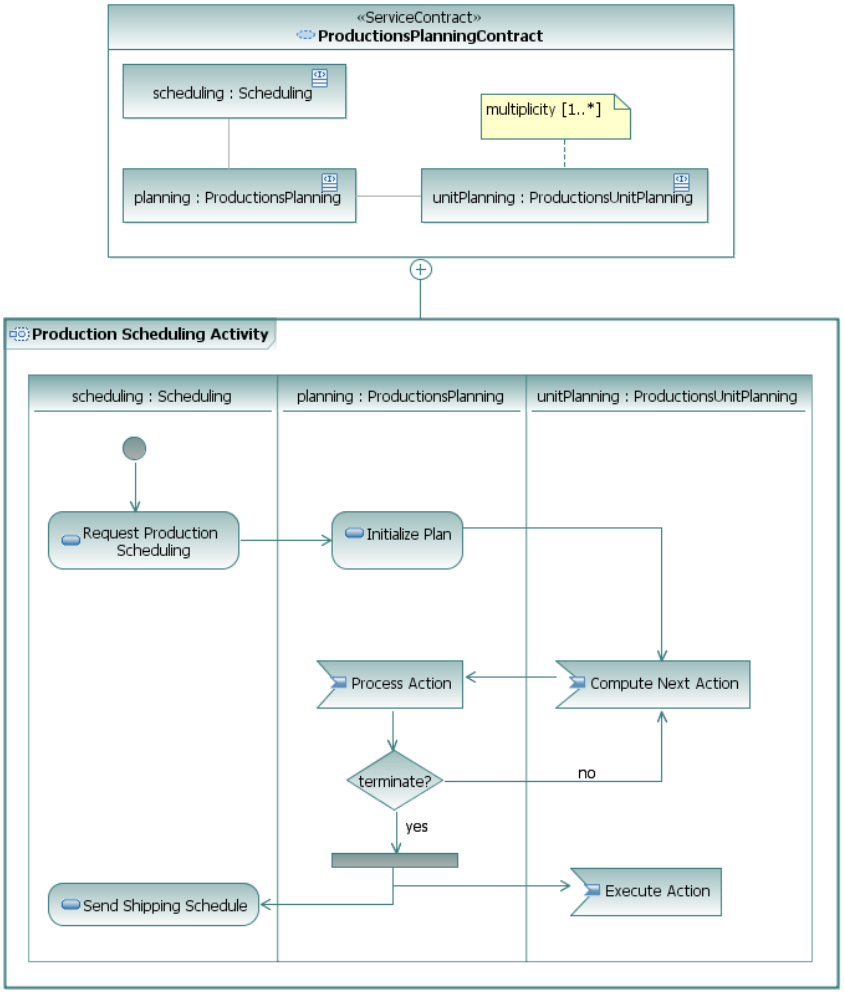


Figure 5.9: *Requirements of the Productions.*

Figure 5.10: *Productions Interfaces Listing Role Responsibilities.*

5.4.2 Service Specification

Figure 5.11 shows a view of the service interfaces determined necessary to fulfill the requirements specified by the collaboration in Figure 5.9. This view has identified new Service Interface, *PlanningService*, which is described in Figure 5.12. The Purchasing service that we described in Figure 5.5 was such a simple service that it was sufficient to describe it with ordinary UML Interface. The PlanningService is little bit more complex and also illustrates the use of ServiceInterfaces within SOA-Pro. The PlanningService provides the ProductionsPlanning interface and requires the ProductionsUnitPlanning interface. In addition it also owns activity diagram which described the collaboration between these two roles (interfaces). The activity is describing high level steps of the STP.

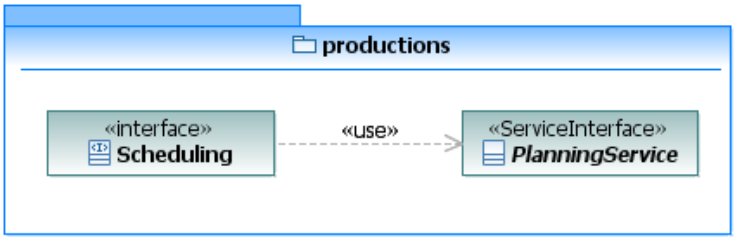


Figure 5.11: *Identified Service Interfaces in Productions.*

From description of case study in Section 2.1, we have relationship between entities as depicted in Figure 5.13

Figure 5.13 depicts a UML class diagram that describes the relations between orders, aggregates, and heats. We use the concept *OrderPositionPart* to express that one *OrderPosition* might be split-up into several parts that are distributed among several *Heats*.

Moreover, each *OrderPositionPart* has to be assigned to an *Aggregate* that fits to the required format and quality constraints of the corresponding OrderPosition. Every Aggregate refers to an ordered list of *Sequences* that represents the local schedule for it (see Figure 2.3). A Sequence consists of an ordered list of Heats that have similar time, quality, and format constraints. The *DTS* is the schedule for the whole steelwork containing the set of all Sequences.

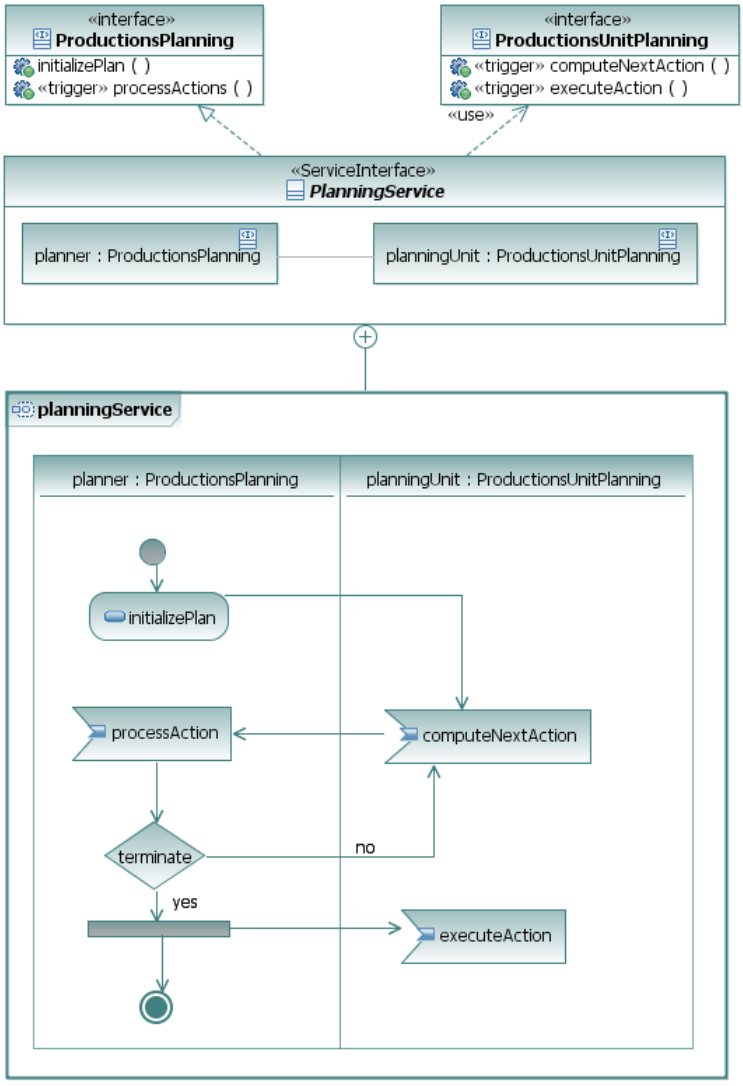


Figure 5.12: *The Planning Service Interface.*

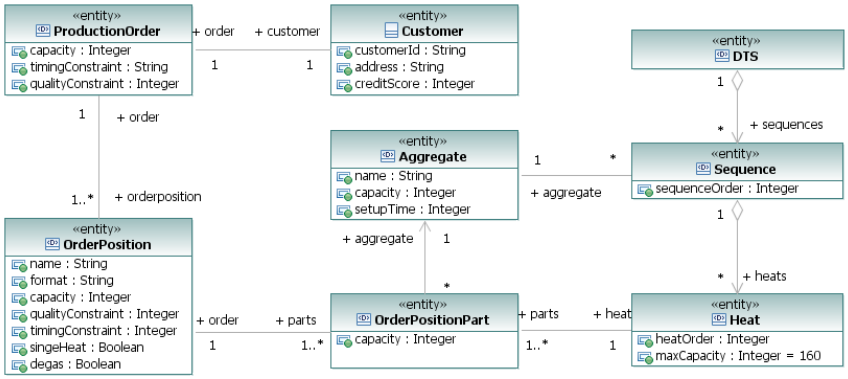


Figure 5.13: *UML class diagram describing the relationship between entities.*

5.4.3 Service Realization

As described earlier in Section 3.4.3, there are two actors in the STP: The *Initiator* initiates and coordinates the execution of the STP, while the *Participants* represent the planning units that decide whether they want to buy or sell something to optimize their local plans. In every round of the STP, the participants send their buy and sell decisions to the initiator that has to decide what actions are executed.

Figure 5.14 depicts the Agent, which is specialization of Participant (see Figure 3.5) in SOA-Pro and is representing the Initiator. The ProductionsPlanner requires the capabilities of the Scheduling interface, provides the ProductionsPlanning and requires PorductionsUnitPlanning interfaces. This is because the Planning service interface is designed from ProductionsPlanner point of view.

Figure 5.14: *The ProductionsPlanner Agent.*

Actor representing the Participant in the STP is depicted in Figure 5.15 and is also stereotyped as Agent. In contrast to ProductionsPlanner, the ProductionsPlanningUnit is requiring the ProductionsPlanning interface and providing the ProductionsUnitPlanning interface.

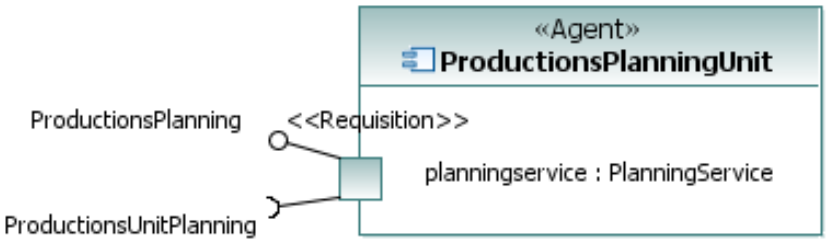


Figure 5.15: *The ProductionsPlanningUnit Agent.*

The Participants in the STP is divided in several subgroups according to the decisions they take in the Buy and Sell Phase. We remember that UML already supports configuration with subsets and the Figure 5.16 depicts how this is done in our use case. We have three UML Collaborations which refines

each other in order to specialize the different roles of the ProductionsPlanningUnits.

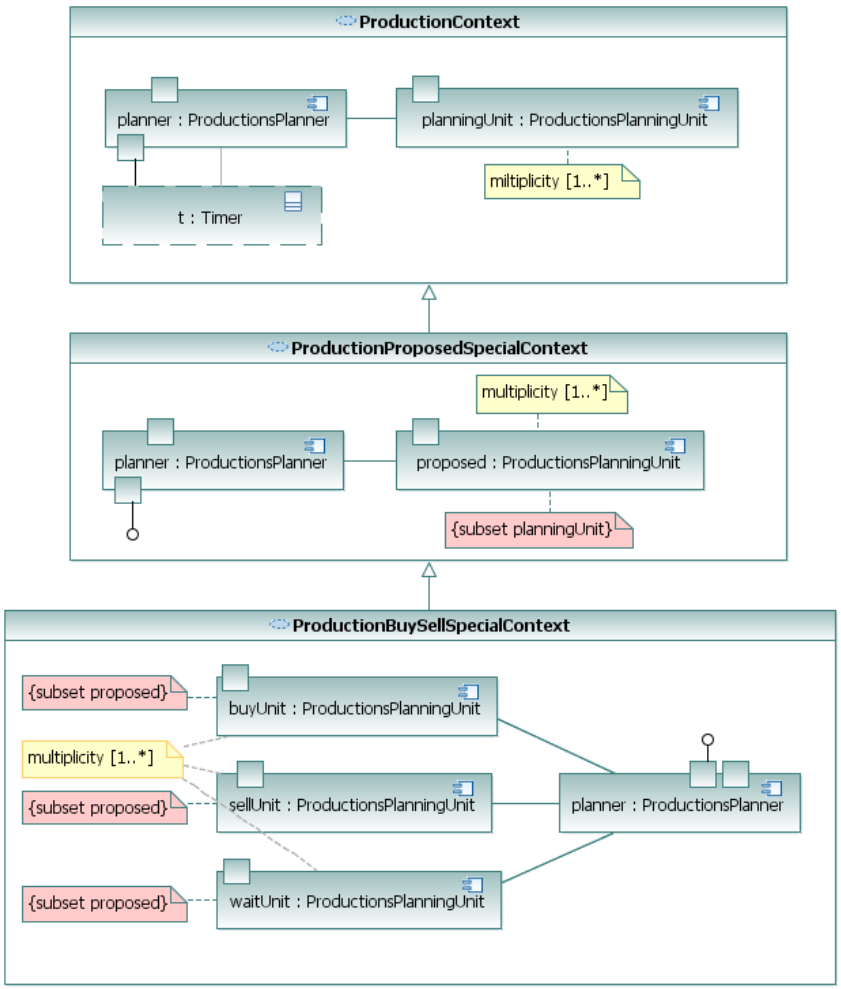


Figure 5.16: *Roles of the simulated trading protocol in Productions.*

We have also included custom Timer class, which is used by ProductionsPlanner for the timer constraints. Operation of the Timer is used to start, stop and read timer actions, as described in Figure 5.17. Details about how these operations are implemented will not be covered here, we only focus on the concepts.

Now that we have identified, specified and realized participants of the STP and the services, the only part that is missing before we can assembling services together is the sequence diagram describing the interaction between participants. This will provide more details to the requirements of the contract from Figure 5.9.

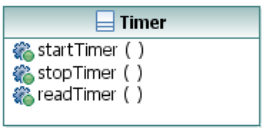


Figure 5.17: *Custom Timer expressed by UML Class.*

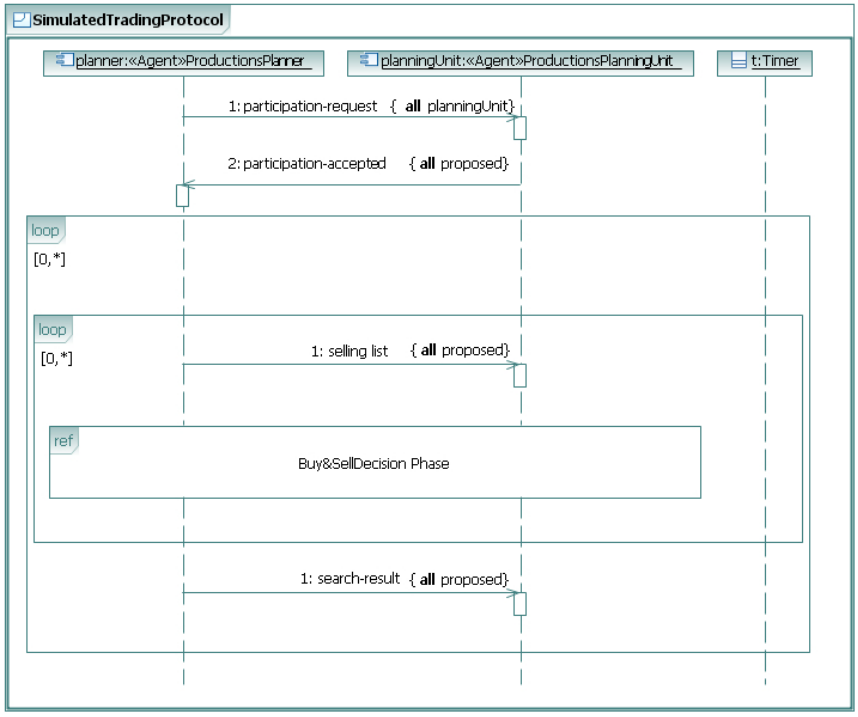


Figure 5.18: *Simulated Trading Protocol in Productions.*

Figure 5.18 depicts the STP, where *planner* starts the protocol with sending the *participation-request* signal to all planningUnits. Then the *Buy&SellDecision Phase* repeats several times before the *search-result* signal is sent to the planningUnits and the protocol starts over again until some threshold. The *Buy&SellDecision Phase* interaction use is described in Figure 5.19. Before the selling and buying interactions starts, the planner calls the *startTimer* operation of the *timer* in order to start the clock. The possible decisions by planningUnits is buy, sell or wait. Last alternative message is the TimeOut signal that could be sent from timer if there is timeout. The different signals that is used is depicted in Figure 5.20. These signals are specializing the FIPA-performatives and we want to use signals instead of method calls since agents are listening to and acting from asynchronous events.

Details about how the planner and planningUnits take their decisions in

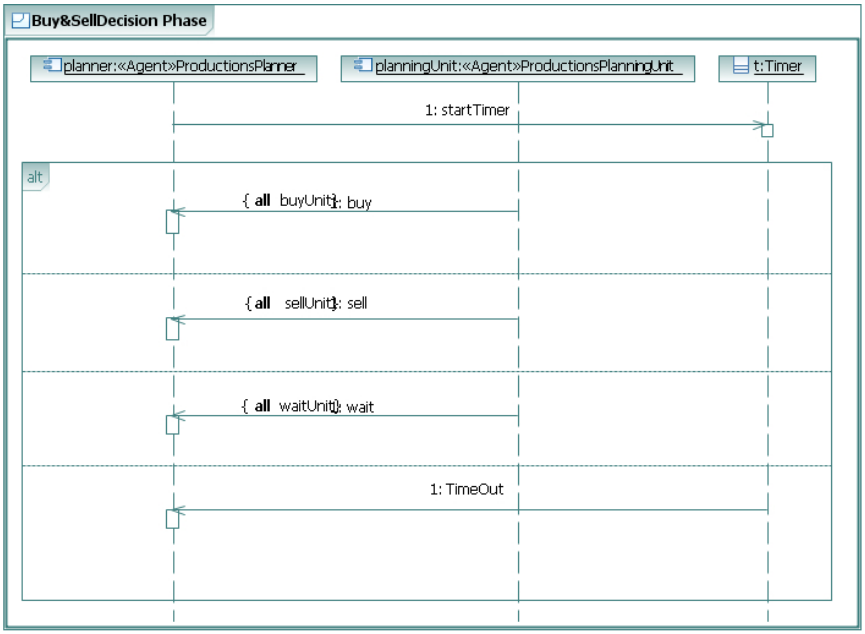


Figure 5.19: *Buy and sell phase of the STP in Productions.*

the STP is left out here because we wanted only to focus on the concepts. Most probably the implementation of how to decide whether to buy/sell or wait in Saarstahl use case, will not be modelled, but rather reused by calling a service which has the implementation already.

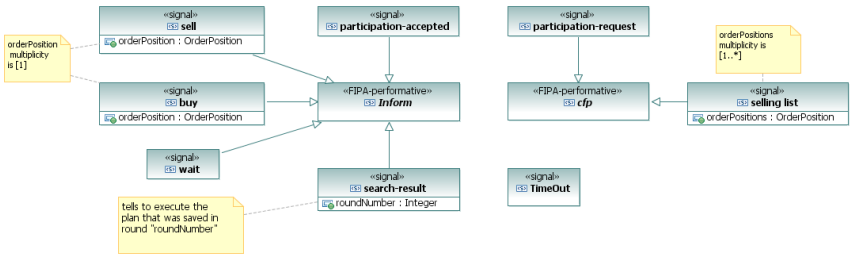


Figure 5.20: *Messages of the STP modelled as UML signals.*

5.4.4 Assembling Services and Fulfilling Contracts

We are now ready to assembly all services together and create the big picture of the Productions participant.

In Figure 5.21, we present how the production part of the PurchaseOrder scenario can be extended to use the STP for production planning. The ProductionsPlannner (stereotype Agent) is the central entity in the STP. It is in

charge of the initialization of the STP and the coordination of the ProductionPlanningUnits (stereotype Agent). Diagram in Figure 5.21 depicts one instance of Agent ProductionPlanner and three instances of the ProductionPlannerUnit. The STP itself is specified in the context of the ProductionsPlanningContract, which defines the message exchange between the involved parties. The instances of ProductionsPlanner perform the Initiator actor of the STP and the instances of ProductionPlanningUnit the Participant role.

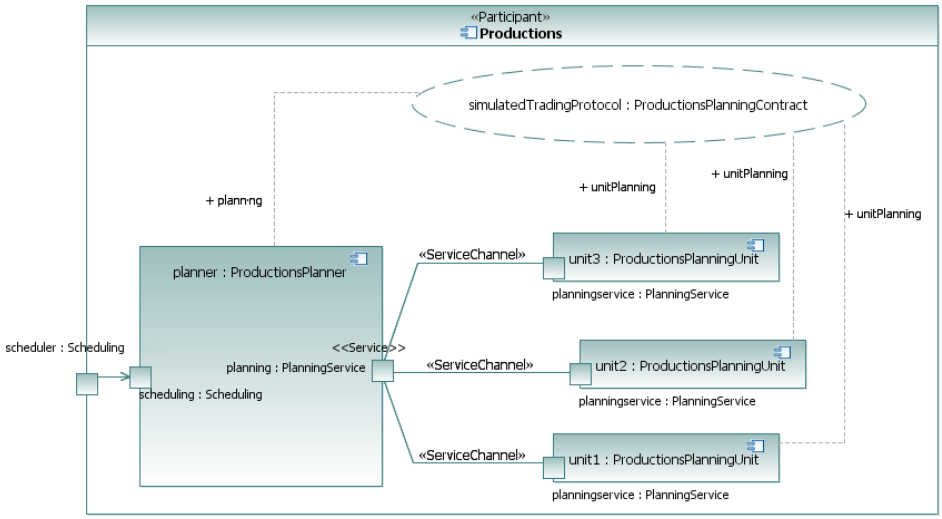


Figure 5.21: *Assembling the parts into a deployable subsystem, Productions.*

5.5 Summary

The realization and implementation chapter explained the work done in the realization of UPMS-a. Concerning the requirements of the case study, it introduced how the purchase order and scheduling production could be modeled in collaboration of services in SOA and interaction protocols in Multi-Agent Systems.

Chapter 6

UPMS-a: Evaluation

> Experience is a hard teacher
> because she gives the test first,
> the lesson afterwards.
>
> Vernon Sanders Law

As we explained in the beginning of this thesis, evaluation will be based on evaluating the hypothesis with respect to the success criteria previously defined in Chapter 2. Every success criterion, is followed by a short argumentation on its validation result.

We validate the success criteria through a level of fulfillment. These levels are **Fulfilled, Partly Fulfilled** and **Poorly Fulfilled**.

6.1 Success criterion 1

A suitable extension of SOA-Pro will make it possible to express multicast of messages between participants in agent interaction protocols.

Fulfilled: UPSM-a introduced subset notation on messages which provided support in UML Sequence Diagrams for expressing which group(s) of participants are sending or receiving particular messages. The implementation of this feature is depicted in case scenario, Figure 5.18.

6.2 Success criterion 2

A suitable extension will make it possible to group participants in groups in order to express which group of participants are receiving or sending particular message.

Fulfilled: UPMS-a presented configuration with subsets in UML, which makes it possible to not only define groups of participants, but also

describe their multiplicity and internal relationships. The implementation of this feature is depicted in case scenario, Figure 5.16.

6.3 Success criterion 3

A suitable extension will only introduce or extend the SOA-Pro proposal where needed, and use as much as possible existing concepts in SOA-Pro or UML.

Fulfilled: UPMS-a has only introduced two new extensions to UML and SOA-Pro; (i) subset notation on messages in Sequence Diagrams and (ii) iterator constructs for multicasting. This was not possible to express with UML 2 since features for multicasting were an lacking feature and the need to express this kind of scenarios have not arise.

6.4 Success criterion 4

A suitable extension will make it clear how to use timer concepts within SOA-Pro in order to express deadlines in interaction protocols.

Fulfilled UPMS-a presented three alternative ways to model timers with UML 2 and SOA-Pro i) custom classifier that represents timer, ii) use of SimpleTime model included in UML Superstructure and iii) more sophisticated model of time provided by an appropriate UML profile. First alternative is implemented in case scenario, Figures 5.17 and 5.19.

6.5 Hypothesis

H1 *Proposed extensions will make SOA-Pro and UML suitable to express agent interaction protocols.*

Through the evaluation of the success criteria, we have validated the hypothesis H1. From the four success criteria that acted as predicates to test the hypothesis, all four were Fulfilled. This result, led us the conclusion that H1 is true. From here we can confirm that the main goal of this thesis is accomplished.

Now we can state that : “Extensions proposed in UPMS-a does make SOA-Pro and UML suitable to express agent interaction protocols."

Chapter 7

Conclusion and Future Work

> Try not to become a man of success, but rather try to become a man of value.
>
> Albert Einstein

The final chapter summarizes this thesis and provides an outlook on feasible future work. First we start with summarizing our work in Section 7.1. Then we emphasize achievements in Section 7.2 and finishing with suggesting feasible future work in Section 7.3.

7.1 Conclusion

In this thesis, we investigated the similarities of Service-Oriented Architectures (SOAs) and Multi-Agent Systems (MASs). For this purpose, we took the revised submission of the UML Profile and Metamodel for Services (SOA-Pro) and evaluated whether the proposed approach supports modeling of the core building blocks of MASs. Our evaluation taught us that SOA-Pro offers basic functionalities to model MASs and thus a model transformation between SOA-Pro and MASs like PIM4Agents is feasible. However, advanced functionalities of MAS, like agent interaction protocols are not supported by SOA-Pro and we showed that SOA-Pro could easily be extended to support these kinds of functionalities. In our evaluation, we mainly concentrated on interaction extensions, however, further extensions are discussed in the ongoing work in the context of the Agent Metamodel and Profile by OMG.

7.2 Achievements

The work with agent-based extensions for the UML Profile for Service-Oriented Architectures:

- identified similarities and needed extensions of SOA-Pro submission in order to collaborate with MASs.
- proposed feasible and small extensions of SOA-Pro and UML 2 in order to model agent interaction protocols.
- modeled one of the very first real world scenarios with SOA-Pro. The submission is still under review and is not yet published for the public. Not only did we present and explain how to use the stereotypes, but we also showed how agents and services can collaborate with each other in the context of MDA.
- has already showed to be of great value for multiple parties; (i) in the ongoing work with AMP RFP, which is research on the same field, but in greater scale, with a number of participants involved and greater time pan. (ii) We have also written a paper [31] that summarize our work and which is to be published for the *Modeling, Design, and Analysis for Service-Oriented Architecture Workshop; 3rd edition* (mda4soa)[1]. (iii) Within SHAPE project our work has already been adopted in the work packages and will be basis for the further research and discussions.

7.3 Future Work

As we mentioned earlier, there is ongoing work in the context of the Agent Metamodel and Profile (AMP) by OMG, which has been in process since we started with our work. We have worked together in defining the requirements in the RFP, which will be covered by submissions.

Basically we believe that first step, in direction of metamodel and profile for Agents, is to make adjustments in SOA-Pro so it supports the core building blocks of MASs that we presented earlier in this thesis (see Section 2.5.

Following, in Section 7.3.1 and 7.3.2 we will suggest future work on respectively implementation of subset notations on messages and definition of the timer concepts, both within AMP. Then we take closer look at two of the core blocks, organization and roles, and suggests how they could be implemented with SOA-Pro. Suggestions we present in Section 7.3.3 and 7.3.4 are just conceptual, and are inspirited by PIM4Agent metamodel. Finally, in Section 7.3.5 we will discuss the future work on the tools for UML and UML Profiles modeling.

7.3.1 Implementation of Subset Notation

We believe that subset notation on messages, as we presented in this thesis, was successful support in order to express multicast of messages within in-

[1] http://events.deri.at/mda4soa2008/

teraction protocols. However, we did only present shorthand notation. Now it's important to take an closer look on how this could be implemented in metamodel and UML Profile for Agents and SOA-Pro.

7.3.2 Definition of Timers

As we described, timer concepts are important with respect to the agent interaction protocols. We suggest that Timer concepts from UML Testing Profile get adopted in AMP and modified to suit UML 2 metamodel. We did try this in our work, but realized that we need more detail knowledge of the UML metamodel, before we can suggest any modifications. Benefit of defining the Timer concepts in the UML Profile, as we suggest here it is easier to start using already existing concepts, and the diagrams gets becomes more compact (as explained in Section 4.5.3).

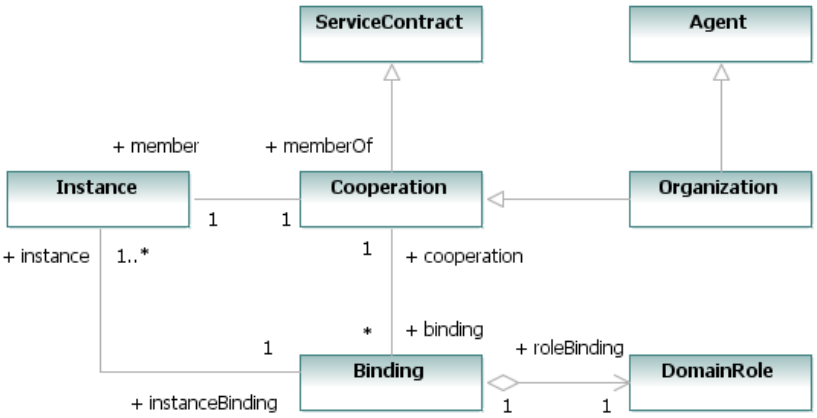


Figure 7.1: *Organizational extensions of SOA-Pro.*

7.3.3 Organizations in SOA-Pro

A *Cooperation* (depicted in Figure 7.1) defines a social structure *Agents* can take part in. It is a specialization of a *ServiceContract* in SOA-Pro and binds *Instances* of *Agent* to *DomainRoles* the *Cooperation* requires. An *Organization* is a special kind of *Cooperation* that also has the characteristics of an *Agent*. Therefore, the *Organization* can perform *DomainRoles* and has *Capabilities*. Using the concepts of an *Organization*, social units can be formed that take advantage of the synergies of its members, resulting in an entity that enables products and processes that are not possible from any single individual.

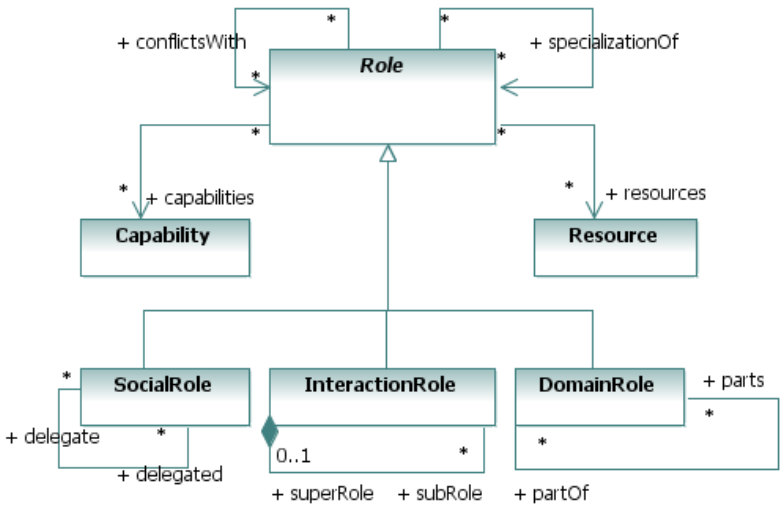


Figure 7.2: *Role extensions of SOA-Pro.*

7.3.4 Roles in SOA-Pro

In general, a *Role* (see Figure 7.2) is an abstraction of the social behavior of an *Agent* in a given social context, usually a *Cooperation*. It refers to (i) a set of *Capabilities* defining the *Behaviors* it can possess and (ii) a set of *Resources* that represent the environment it has access to. Three specializations of a *Role* are considered in Figure 7.2. A *DomainRole* gives an *Agent* certain *Capabilities* inside a particular domain whereas an *InteractionRole* defines the entities within *Interactions* and their capabilities to behave in accordance to the *Interaction* with respect to sending and receiving messages. *InteractionRoles* can also be split using the subactor reference meaning that the *Instances* that are bound to the superactor can be part of exactly one subactor. In general, the subactors are filled at run-time, however, the range of *Instances* can be defined at design time using the min and max attributes. Beside the *DomainRole* and *InteractionRole*, a *SocialRole* defines which responsibilities it needs to fulfill in the particular social unit it is acting.

7.3.5 Complete UML 2 Tool Support

As we could see in our comparison of the modeling tools for UML in Table 2.4, there is poor support for complete UML modeling. We found IBM RSM

7.0.5 as the best alternative, among several other commercial and open-source tools. However, neither of the tools did completely support UML 2, and if the MDA approach should be successfully used in the industry, it is required that the tool support gets much better and with advantage on the open-source frameworks like Eclipse.

Appendix A

List of Abbreviations

AOSE	Agent-Oriented Software Engineering
BDI	Belief, Desire, Intention
CIM	Computational Independent Model
AI	Artificial Intelligence.
DAI	Distributed Artificial Intelligence
DSL	Domain Specific Language
MAS	Multi-Agent System
MDA	Model-Driven Architecture
MDSD	Model-Driven Software Development
MDD	Model-Driven Development
MOF	Meta Object Facility
PIM	Platform-Independent Model
PSM	Platform-Specific Model
SOA	Service-Oriented Architecture
UML	Unified Modeling Language
CNP	Contract Net Protocol
STP	Simulated Trading Protocol
PDP	Pickup and Delivery Problem
CAL	Communicative Act Library
FIPA	Foundation for Intelligent Physical Agents
ACL	Agent Communication Language
WS	Web Service
WSA	Web Service Architecture
JADE	Java Agent DEvelopment Framework
AUML	Agent UML
OMG	Object Management Group
OASIS	Organization for the Advancement of Structured Information Standards
SF	Software Factories
DSM	Domain Specific Modeling

Bibliography

[1] Bachem A., Hochstättler W., and Malich M. *Simulated Trading: A New Parallel Approach for Solving Vehicle Routing Problem* Report No. 92.125, 1992.

[2] Chavez A. and Maes P. *Kasbah: An agent marketplace for buying and selling goods*. In *First International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM'96)*, pages 75–90, London, UK, 1996. Practical Application Company.

[3] AAMAS. *Autonomous Agents and Multiagent Systems. 4th workshop on agents in traffic and transportation.* [online]. Available from: http://ki.informatik.uni-wuerzburg.de/~kluegl/att2006/ [cited 4. June 2008].

[4] Ali Arsanjani and Abdul Allam. *Service-Oriented Modeling and Architecture for Realization of an SOA*. In *SCC '06: Proceedings of the IEEE International Conference on Services Computing*, Washington, DC, USA, 2006. IEEE Computer Society.

[5] Krulwich B. *Bargain finder agent prototype*, Anderson Consulting, 1995.

[6] G. Benguria, X. Larrucea, B. Elvesæter, T. Neple, A. Beardsmore, and M. Friess. *A Platform Independent Model for Service Oriented Architectures*. In *Second International Conference on Interoperability of Enterprise Software and Applications (I-ESA 2006)*, 2006.

[7] Bauer Bernhard and Odell James. *UML 2.0 and Agents: How to Build Agent-based Systems with the new UML Standard*. *Journal of Engineering Applications of Artificial Intelligence*, 18(2):141–157, March 2005.

[8] Jeffrey M. Bradshaw. *An Introduction to Software Agents*. In Jeffrey M. Bradshaw, editor, *Software Agents*, pages 3–46. AAAI Press / The MIT Press, 1997.

[9] M. Bratman. *Intentions, Plans, and Practical Reason*. Harvard University Press, Cambridge, Massachusetts, 1987.

[10] P. Bresciani, P. Giorgini, F. Giunchiglia, J. Mylopoulos, and A. Perini. *Tropos: An Agent-Oriented Software Development Methodology*. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, 2004.

[11] R. Brooks. *A robust layered control system for a mobile robot*. *Robotics and Automation, IEEE Journal of [legacy, pre - 1988]*, 2(1):14–23, 1986.

[12] Hahn Christian. *A Platform Independent Agent-based Modeling Language*, German Research Center for Artificial Intelligence (DFKI), 2008.

[13] D. E. Comer, David Gries, Michael C. Mulder, Allen Tucker, A. Joe Turner, and Paul R. Young. *Computing as a discipline*. *Communications of the ACM*, 32(1):9–23, 1989.

[14] SHAPE Consortium. *SHAPE Project* [online]. Available from: http://www.shape-project.eu/ [cited 02. June 2008].

[15] University of Hamburg Department of Informatics, MIN Faculty. *JADEX: BDI Agent System* [online]. Available from: http://vsis-www.informatik.uni-hamburg.de/projects/jadex/ [cited 16. June 2008].

[16] E. H. Durfee. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. MIT Press, March 1999.

[17] IBM et al. *UML Profile and Metamodel for Services (SOA-Pro)* [online]. Available from: http://www.omg.org/docs/ad/08-05-03.pdf [cited 01. July 2008].

[18] FIPA. *FIPA Agent Communication Language Specifications* [online]. Available from: http://www.fipa.org/repository/aclspecs.html [cited 02. June 2008].

[19] FIPA. *IEEE Foundation for Intelligent Physical Agents. The foundation for intelligent physical agents* [online]. Available from: http://www.fipa.org/ [cited 02. June 2008].

[20] FIPA. *FIPA Communicative Act Library Specification*, Foundation for Intelligent Physical Agents, 2002.

[21] FIPA. *FIPA Contract Net Interaction Protocol Specification*, Foundation for Intelligent Physical Agents, 2002.

[22] K. Fischer, C. Hahn, and C. Madrigal-Mora. *Agent-oriented software engineering: a model-driven approach*. *International Journal of Agent-Oriented Software Engineering*, 1(3/4), 2007.

[23] The Eclipse Foundation and SINTEF. *MOFScript Home page* [online]. 2007. Available from: http://www.eclipse.org/gmt/mofscript/ [cited 05. July 2008].

[24] Smith Reid G. and Davis R. *The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver*. *IEEE TRANSACTIONS ON COMPUTERS*, C-29(12), 1980.

[25] gameai.com. *AI in Games: A Personal View* [online]. Available from: http://www.gameai.com/blackandwhite.html [cited 19. June 2008].

[26] Francisco J. Garijo and Magnus Boman, editors. *MultiAgent System Engineering, 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW '99, Valencia, Spain, June 30 - July 2, 1999, Proceedings*, volume 1647 of *Lecture Notes in Computer Science*. Springer, 1999.

[27] Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, August 2004.

[28] Agent Oriented Software Group. *JACK* [online]. Available from: http://www.agent-software.com/jack.html [cited 16. June 2008].

[29] W3C Web Services Architecture Working Group. *Web Service Architecture* [online]. February 2004. Available from: http://www.w3.org/TR/ws-arch/ [cited 12. June 2008].

[30] Christian Hahn, Cristian Madrigal-Mora, Klaus Fischer, Brian Elvesæter, Arne-J370rgen Berre, and Ingo Zinnikus. *Meta-models, Models, and Model Transformations: Towards Interoperable Agents*. In *Proceedings of the 4th German Conference on Multiagent System Technologies*, volume 4196 of *Lecture Notes in Computer Science*, Berlin/Heidelberg, 2006. Springer.

[31] Christian Hahn and Ismar Slomic. *Agent-based Extensions for the UML Profile and Metamodel for Service-oriented Architectures*. Saarbrücken, Germany and Oslo, Norway, 2008. IEEE Computer Society (To be published).

[32] Bürckert Hans-Jürgen, Fischer Klaus, and Vierke Gero. *Holonic transportation scheduling with teletruck*. *Journal of Applied Artificial Intelligence*, 2000.

[33] Solheim Ida and Stølen Ketil. *Technology Research Explained* SINTEF A313, SINTEF, March 2006.

[34] Odell James J. *Objects and Agents Compared*. *Journal of Object Technology*, 1(1):41–53, May/June 2002.

[35] Russell Stuart J. and Norvig Peter. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003. Available from: http://portal.acm.org/citation.cfm?id=773294.

[36] Runde R. K. *STAIRS - Understanding and Developing Specifications Expressed as UML Interaction Diagrams*. In *Department of Informatics*, page 348. University of Oslo, 2007.

[37] Fischer Klaus, Müller Jörg P., and Pischel Markus. *Cooperative Transportation Scheduling: an application Domain for (DAI)* RR-95-01, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH, Kaiserslautern, Germany, 1995.

[38] Fischer Klaus, Müller Jörg P., and Pischel Markus. *Cooperative transportation scheduling: An application domain for dai*, Computational Economics, 1996.

[39] Fischer Klaus, Muller J. P., Pischel Markus, and Schier Darius. *A Model for Cooperative Transportation Scheduling*. In *Proceedings of the First International Conference on Multiagent Systems.*, pages 109–116, Menlo park, California, June 1995. AAAI Press / MIT Press.

[40] Runde Ragnhild Kobro, Haugen Øystein, and Stølen Ketil. *The Pragmatics of STAIRS*. In *Formal Methods for Components and Objects*, volume 4111, pages 88–114. Springer, 2006.

[41] Helouet L. *Distributed system requirement modeling with message sequence charts: the case of the RMTP2 protocol*. *Information and Software Technologys*, 45(11):701–714, August 2003.

[42] Telecom Italia Lab. *Java Agent DEvelopment Framework* [online]. Available from: http://jade.tilab.com/ [cited 19. June 2008].

[43] ATLAS Group (INRIA & LINA). *ATL (Atlas TransformationLanguage)* [online]. 2005. Available from: http://www.eclipse.org/m2m/atl/ [cited 05. July 2008].

[44] Magnus Ljungberg and Andrew Lucas. *The (OASIS) air-traffic management system*. In *Proceedings of the Second Pacific Rim International Conference on Artificial Intelligence (PRICAI '92)*, Seoul, Korea, 1992.

[45] Huget M.-P. *Extending Agent UML Sequence Diagrams*, volume 2585/2003 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2003. ISBN 978-3-540-00713-5.

[46] Eric A. Marks and Michael Bell. *Service-Oriented Architecture (SOA): A Planning and Implementation Guide for Business and Technology.* John Wiley & Sons, Inc., New York, NY, USA, April 2006.

[47] Wooldridge Michael. *Introduction to MultiAgent Systems.* John Wiley & Sons, June 2002.

[48] Wooldridge Michael and Dickinson I. *Agents are not (just) web services: considering BDI agents and web services*, HP Laboratories Bristol, July 2005. HPL-2005-123.

[49] Wooldridge Michael and Jennings Nicholas R. *Intelligent Agents: Theory and Practice. Knowledge Engineering Review*, 10(2):115–152, 1995.

[50] Wooldridge Michael, Jennings Nicholas R., and Kinny David. *The Gaia Methodology for Agent-Oriented Analysis and Design. Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, 2000.

[51] Jishnu Mukerji and Joaquin Miller. *MDA Guide Version 1.0.1* [online]. 2003. Available from: http://www.omg.org/docs/omg/03-06-01.pdf [cited 03. July 2008].

[52] Jennings N. and Wooldridge Michael. *Agent-Oriented Software Engineering*. In Francisco J. Garijo and Magnus Boman, editors, *Proceedings of the 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World : Multi-Agent System Engineering (MAAMAW'99)*, volume 1647, pages 1–7. Springer-Verlag: Heidelberg, Germany, June 30 - July 2 1999.

[53] OASIS. *Reference Model for Service Oriented Architecture 1.0* [online]. October 2006. Available from: http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.html [cited 01. July 2008].

[54] OMG. *Agent Metamodel and Profile (AMP), Request For Proposal* [online]. Available from: http://www.omg.org/docs/ad/08-06-02.pdf [cited 21. June 2008].

[55] OMG. *Catalog of UML Profile Specifications* [online]. Available from: http://www.omg.org/technology/documents/profile_catalog.htm [cited 05. July 2008].

[56] OMG. *UML Profile and Metamodel for Services (UPMS), Request For Proposal* [online]. Available from: http://www.omg.org/cgi-bin/doc?soa/06-09-09 [cited 03. July 2008].

[57] OMG. *UML Testing Profile V1.0* [online]. Available from: http://www.omg.org/cgi-bin/apps/doc?formal/05-07-07.pdf [cited 27. June 2008].

[58] OMG. *UML: Unified Modeling Language* [online]. Available from: http://www.uml.org [cited 05. July 2008].

[59] OMG. *Meta Object Facility (MOF) Core Specification, OMG Available Specification Version 2.0* [online]. 2006. Available from: http://www.omg.org/spec/MOF/2.0/PDF [cited 05. July 2008].

[60] OMG. *OMG Unified Modeling Language (OMG UML): Infrastructure, V2.1.2* [online]. 2007. Available from: http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF [cited 05. July 2008].

[61] OMG. *OMG Unified Modeling Language (OMG UML): Superstructure V2.1.2* [online]. 2007. Available from: http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF [cited 22. June 2008].

[62] Lin Padgham and Michael Winikoff. *Developing Intelligent Agent Systems: a practical guide*. John Wiley and Sons, 2004.

[63] M. Papasimeon and C. Heinze. *Extending the UML for designing JACK agents*. In *Proceedings of the Australian Software Engineering Conference (ASWEC 01)*, 2001.

[64] Payne Terry R. *Web Services from an Agent Perspective*. 23(2), 2008.

[65] Rao Anand S. and Georgeff Michael P. *Modeling Rational Agents within a BDI-Architecture*. In James Allen, Richard Fikes, and Erik Sandewall, editors, *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, pages 473–484. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA, 1991.

[66] Lund Mass Soldal. *Operational analysis of sequence diagram specification*. PhD thesis, Faculty of Mathematics and Natural Sciences, University of Oslo, 2008.

[67] Warwas Stefan and Hahn Christian. *The Concrete Syntax of the Platform Independent Modeling Language for Multiagent Systems*, DFKI GmbH, 2008.

[68] Ming Tan. *Multi-Agent Reinforcement Learning: Independent vs. Cooperative Learning*. In Michael N. Huhns and Munindar P. Singh, editors, *Readings in Agents*, pages 487–494. Morgan Kaufmann, San Francisco, CA, USA, 1997.

[69] Whitestein Technologies. *Autonomic Business Solutions* [online]. Available from: http://www.whitestein.com/autonomic-business-solutions [cited 11. June 2006].

[70] Haugen Øystein. *Challanges to UML 2 to Describe FIPA Agent Protocol*, 2008.

[71] Haugen Øystein and et al. *STAIRS towards formal design with sequence diagrams. Software and System Modeling (SoSyM)*, 4(4):355–367, 2005.

[72] Haugen Øystein and Stølen Ketil. *STAIRS - Steps To Analyze Interactions with Refinement Semantics*. In *UML 2003*. Springer-Verlag, 2003.

[73] Haugen Øystein and Pedersen B. Møller. *Configurations by UML*. In *European Workshop on Software Architecture*, volume 4344/2006, pages 98–112. Springer, 2006.

[74] Ingo Zinnikus, Christian Hahn, Michael Klein, and Klaus Fischer. *An Agent-Based, Model-Driven Approach for Enabling Interoperability in the Area of Multi-brand Vehicle Configuration*. In *Proc. of the Fifth International Conference on Service-Oriented Computing (ICSOC)*, volume 4749 of *Lecture Notes in Computer Science*, pages 330–341. Springer Verlag, 2007.